6-22-2017 12:00 AM

Resource Bound Guarantees via Programming Languages

Michael J. Burrell
The University of Western Ontario

Supervisor
Mark Daley
The University of Western Ontario Joint Supervisor
James Andrews
The University of Western Ontario

Graduate Program in Computer Science
A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy

Abstract

We present a programming language in which every well-typed program halts in time polynomial with respect to its input and, more importantly, in which upper bounds on resource requirements can be inferred with certainty.

Ensuring that software meets its resource constraints is important in a number of domains, most prominently in hard real-time systems and safety critical systems where failing to meet its time constraints can result in catastrophic failure. The use of testing in ensuring resource constraints is of limited use since the testing of every input or environment is impossible in general. Static analysis, whether via the compiler or complementary programming tool, can generate proofs of correctness with certainty at the cost that not all programs can be analysed.

We describe a programming language, Pola, which provides upper bounds on resource usage for well-typed programs. Further, we describe novel features of Pola that make it more expressive than existing resource-constrained programming languages.

Keywords: programming language, static analysis, resource bounds, type inference, polynomial time, time complexity, category theory.

Acknowlegements

I would like to acknowledge everyone who has helped push this thesis towards completion. It is with a great sense of achievement that I can finally present a unique and worthwhile contribution.

I would like to first thank my supervisors, Mark Daley and Jamie Andrews. Jamie was invaluable in checking every example and every logical sequent. Without him, the meat of the thesis, the middle chapters, would be a mess of abbreviated, inconsistent and unreadable syntax. Mark provided initial motivations, an eye for proofs and what needs to be more formalized and more thoroughly explained, and last-minute proofreading and logistical support.

I would also like to give sincere thanks to my examiners Amy Felty, Lucian Ilie, Marc Moreno Maza, and David Jeffrey, all four of whom provided insightful and provoking questions during the defence that will guide the future development of Pola.

Much of the work developed out of late-night sessions with Robin Cockett and Brian Redmond. They were the main drivers in breaking things, pointing out strange properties, and the last line of defence in ensuring the whole thing worked. Brian did more than his fair share of proofreading papers, as well. I am highly grateful to them both.

I am also grateful to Franziska Biegler, who started off as an office-mate, turned into a fantastic friend and personal supporter, and who did remkarkable amounts of proofreading in great detail.

I cannot help but thank those who gently—and sometimes less than gently—pushed me to "just finish it already". I can say without any hint of glibness that the thesis could never have been finished without well-wishers prodding me into the last machinations of the thesis. These range from my Associate Dean at Sheridan, Philip Stubbs, to my close friend, Dan Santoni, to my immediate family, to my wife, Sinea, who was successful in giving the most welcomed deadline.

Contents

Abstract **ii**

List of Figures **vi**

List of Tables **ix**

1 Introduction **1**
1.1 Functional languages 1
1.2 Implementation 3
1.3 Summary 3

2 Literature survey **5**
2.1 Abstract interpretation 5
2.2 Restricted models of computation 6
2.2.1 Primitive recursion 6
Limited recursion and the Grzegorczyk hierarchy 7
Predicting time complexity of recursive functions 9
Limitations to primitive recursion 9
2.2.2 Hofmann survey 9
Loop programs 10
2.2.3 The Hume languages 11
Hardware level 11
Finite state machine 11
Templates 12
Primitive recursion 13
2.2.4 Categorical approaches 13
Paramorphisms 13
Charity 15
2.3 Type theoretic approaches to determining resource consumption 16
2.3.1 Stack language 16
2.3.2 Memory bounds 17
2.4 Static memory management 19
2.5 Conclusion 20

3 Compositional Pola **21**
3.1 Introductory examples 21

3.1.1 Recursion-free examples . . . 21
3.1.2 Inductive data types and folds . . . 22
Typing restrictions on folds . . . 24
3.1.3 Coinductive data types . . . 26
3.1.4 Unfolds . . . 27
3.1.5 Cases and peeks . . . 28
3.2 Types . . . 29
3.2.1 Syntax of type expressions . . . 29
3.2.2 Syntax of type definitions . . . 30
3.2.3 Constructor and destructor typing . . . 32
3.3 Terms . . . 32
3.3.1 Syntactic sugar . . . 34
3.4 Typing and well-typedness . . . 35
3.4.1 Contexts and sequents . . . 35
3.4.2 Simple terms . . . 36
3.4.3 Recursive inductive terms . . . 36
3.4.4 Coinductive terms . . . 42
3.4.5 Summary . . . 47
3.5 Operational semantics . . . 47
3.6 Type inference . . . 51
3.6.1 Sequents . . . 51
3.6.2 Inductive terms . . . 52
3.6.3 Coinductive terms . . . 62
3.6.4 Unification . . . 65
Substitution and matching . . . 66
3.6.5 Correctness of type inference . . . 71

4 Pola implementation **73**

5 Bounds inference **74**
5.1 Sizes . . . 74
5.1.1 Operations on sizes . . . 75
Addition . . . 76
Subtraction . . . 77
Multiplication . . . 77
Maximum . . . 77
Dot product . . . 78
Application . . . 78
Destructor application . . . 78
5.1.2 Inferring size bounds . . . 80
Folds . . . 80
Non-recursive coinductive sizes . . . 87
Recursive coinductive sizes . . . 90
5.1.3 Correctness of inference . . . 94
Defining the potentially recursive size of a term, $\overset{\square}{\bowtie}$ . . . 94

Definitions of inferred size bounds, $\overset{\star}{\bowtie}$ 96
Relationship between potential recursive size and inferred size bound 97
5.2 Times 100
5.2.1 Times 101
Operations on times 101
5.2.2 Environments 102
5.2.3 Sequents 102
5.2.4 Inductive terms 103
5.2.5 Coinductive terms 108
5.2.6 Correctness of inference 108
5.3 Polynomial time constraint 113

6 Expressiveness **116**
6.1 Expressing simple functions 116
6.2 Limits 122
6.3 Variations of Pola 122
6.3.1 An elementary-recursive language 122
6.3.2 A PSPACE language 125
6.3.3 A more expressive P-complete language 128

7 Conclusion **131**

Bibliography **131**

Curriculum Vitae **134**

List of Figures

1.1 A Java method which sums the elements of an array of integers and returns that sum. . . . 2
1.2 A Haskell function which sums the elements of a list of integers and returns that sum. . . . 2
1.3 A Pola function which sums the elements of a list of integers and returns that sum. . . . 2

2.1 A finite state machine to decide if a binary number is divisible by 3. . . . 12

3.1 A demonstration of the distinction between opponent and player variables. The function f is legal in Pola, but the function g is illegal. . . . 21
3.2 An example of inductive data types and folds in Pola. . . . 22
3.3 Addition and multiplication functions defined in Pola. . . . 24
3.4 A failed attempt at writing an exponential function in Pola. This example will lead to a typing error. . . . 25
3.5 An exponential function which would yield a typing error in Pola. . . . 25
3.6 An example of a coinductive data type in Pola, representing the product of two values. . . . 26
3.7 An example of a coinductive data type in Pola. . . . 27
3.8 Infinite lists in Pola. . . . 27
3.9 Examples of using the **case** and **peek** constructs. . . . 28
3.10 Type expressions in Compositional Pola. . . . 29
3.11 Abstract syntax for type definitions in Compositional Pola. . . . 30
3.12 Example inductive data types in Pola. . . . 31
3.13 Example coinductive data types in Pola. . . . 31
3.14 Syntax for terms in Compositional Pola. . . . 33
3.15 An example of Compositional Pola syntax, defining two functions, *add* and *append*, which operate on inductive data. . . . 34
3.16 An example of Compositional Pola syntax, defining two functions, *allNats* and *map*, which operate on coinductive data. . . . 34
3.17 Typing rules for simple terms. . . . 37
3.18 Typing rules for simple branches and pattern-matching. . . . 38
3.19 Typing rules for inductive recursive terms. . . . 38
3.20 The *parity* function, which determines the parity of the natural number x, and its auxiliary function *not*, both given in both Pola and Compositional Pola syntaxes. . . . 39
3.21 A continuation of the derivation below. . . . 40

3.22 A continuation of the derivation below. 40
3.23 A continuation of the below derivation. 41
3.24 A type derivation of the *parity* function given in figure 3.20. 41
3.25 Typing rules for coinductive terms. 43
3.26 The *allNats* function, which produces the infinite list of all natural numbers. 44
3.27 A continuation of the below derivation. 45
3.28 A continuation of the below derivation. 45
3.29 A type derivation of the function *allNats* given in figure 3.26. 46
3.30 A hypothetical exponential function which does not rely on duplication of player variables. *expX* would compute the value 2^x for any natural number x if it were able to be typed. 48
3.31 Operational semantics for inductive terms in compositional Pola. 49
3.32 Operational semantics for coinductive terms in compositional Pola. . . . 50
3.33 Basic rules of type inference. 53
3.34 Rules of type inference for pattern matching. 54
3.35 Rules for halting pattern matching. 55
3.36 Rules of type inference for tuples. 55
3.37 Rules of type inference for **fold** constructs. 56
3.38 A continuation of the below derivation. 57
3.39 A continuation of the below derivation. 58
3.40 A continuation of the below derivation. 59
3.41 A continuation of the below derivation. 60
3.42 A continuation of the below derivation. 60
3.43 Type inference of the *parity* function given in figure 3.20. 61
3.44 Rules of inference for non-recursive coinductive terms. 62
3.45 Rules of inference for recursive coinductive terms. 63
3.46 A continuation of the derivation below. 64
3.47 A continuation of the derivation below. 64
3.48 Type inference on the *allNats* function given in figure 3.26. 64

5.1 A short compositional Pola function which prefixes its argument, a list of natural numbers, with the number 0. 75
5.2 Basic rules of size inference. 79
5.3 Rules of size inference for case constructs. 81
5.4 Rules of size inference for **fold** constructs. 82
5.5 A continuation of the derivation given in figure 5.6. 84
5.6 A continuation of the derivation given in figure 5.7. 84
5.7 A continuation of the derivation given in figure 5.8. 84
5.8 A continuation of the derivation given in figure 5.11. 85
5.9 A continuation of the derivation given in figure 5.10. 85
5.10 A continuation of the derivation given in figure 5.11. 86
5.11 Size bound inference for the *add* function given in figure 3.3. 86
5.12 Rules of size inference for non-recursive coinductive size bounds. 87
5.13 Size bound inference on the non-recursive coinductive Pola term (Eval : $x.\mathsf{Succ}(x)$). 89

5.14 Rules of size inference for recursive coinductive size bounds. 91
5.15 A continuation of the derivation given in figure 5.17. 92
5.16 A continuation of the derivation below. 92
5.17 Size inference on the *allNats* function. 93
5.18 A brief summary of each of the 10 definitions introduced in this section. . 97
5.19 Time bounds for simple Compositional Pola terms. 104
5.20 Time bounds for inductive constructs in Compositional Pola. 105
5.21 A continuation of the derivation below. 106
5.22 A continuation of the derivation below. 106
5.23 A continuation of the below derivation. 106
5.24 Time bound inference for the *add* function given is figure 3.3. 107
5.25 Time bounds for coinductive terms. 109
5.26 Potential time bounds for simple Compositional Pola terms. 110
5.27 Potential time bounds for inductive constructs in Compositional Pola. . . 111
5.28 A simulation of a Deterministic Turing Machine in Pola. This depends on the functions `step`, `accepting` and `q` to be defined appropriately for the given Turing Machine. 115

6.1 An implementation of the less-than-or-equal-to function, *leq*, in Compositional Pola and Pola. 117
6.2 A continuation of the derivation given below. 118
6.3 A continuation of the derivation given below. 118
6.4 Continuation of the derivation given below. 119
6.5 Continuation of the derivation given below. 119
6.6 A continuation of the derivation given below. 120
6.7 A continuation of the derivation given below. 120
6.8 Time bound inference for the *leq* function given in figure 6.1. 121
6.9 An implementation of the insert function, *insert*, in Compositional Pola and Pola. 122
6.10 An implementation of the insertion sort function, *insertionSort*, in Compositional Pola and Pola. 123
6.11 A comparison of type inference rules for the **peek** construct in Compositional Pola (top) and Affine-Free Compositional Pola (bottom). 123
6.12 An example of exponential-time behaviour in Affine-Free Compositional Pola. 124
6.13 A solution to the Quantified Boolean Formula problem, given in Exploratory Compositional Pola. 126
6.14 The simplified typing rules of the **peek** construct in Compositional Pola (top) and Dangerous Compositional Pola (bottom). The introduction of variables of type α is not given in these rules, for brevity. 128
6.15 An example of how to generate an exponential-time function if safe and dangerous type restrictions are not put on records. 129
6.16 Modified typing rules for records in Dangerous Compositional Pola. . . . 129
6.17 A second example of an exponential-time function if safe and dangerous type restrictions are not properly put on records. 129

List of Tables

5.1 The three forms of sequents used during time bound inference. 103

Chapter 1

Introduction

Nearly every programming language in common use today—for instance, C, Java, Lisp, PHP, Perl, Javascript, Matlab—is universally powerful. That is, anything which can be computed can be computed using these languages[1]. This is a benefit in allowing an expressive and natural style of programming, and in allowing computationally intensive functions, such as exponential-time functions, to be written, but it is an impediment to static analysis. It is a clear consequence of the Church-Turing thesis that one cannot provide a method which, in general, will determine the running time of a function. This thesis will explore inferring running times, but will focus on a computational model which is not universally powerful and hence to which the Church-Turing thesis is not applicable.

This thesis describes the programming language Pola, a programming language which is not universally powerful. Specifically, it is limited to the polynomial-time functions: it is impossible in Pola to write a function which does not halt in time polynomial with respect to the size of its input. We will then show how this restriction placed on Pola allows the practical bounds of running times of functions written in the language to be automatically inferred by a Pola interpreter or compiler.

The primary applications of a language like Pola can most clearly be seen in hard real-time software systems, such as medical devices, automotive controls or industrial controllers. These systems consider any software component which does not meet its specified time constraints to be a system failure. Having an automated guaranteed bound on running time from a compiler would relieve the burden from the programmer and would be more reliable than testing to ensure the software meets its specifications. Since Pola is not universally powerful, it would likely never comprise the totality of a software system, but could provide the primary CPU-bound components in conjunction with other software components.

1.1 Functional languages

Pola is a functional programming language. The two hallmarks of functional programming are: firstly, that functional programs are declarative, in that functions consist of expressions to evaluate, rather than featuring a sequence of statements or instructions

[1]This ignores the technicality that on a physical computer one is bound by a fixed amount of memory.

```
1 | public static int sumArray(int[] xs) {
2 |     int sum = 0;
3 |     for (int i = 0; i < xs.length; i++)
4 |         sum += xs[i];
5 |     return sum;
6 | }
```

Figure 1.1: A Java method which sums the elements of an array of integers and returns that sum.

```
1 | sumArray [] = 0
2 | sumArray (x : xs) = x + sumArray xs
```

Figure 1.2: A Haskell function which sums the elements of a list of integers and returns that sum.

to execute, as in an imperative language; and secondly, that functional programs do not allow for side-effects such as global variables or direct access to memory locations. The second hallmark mentioned allows what is sometimes called *referential transparency*, wherein a term or subterm may be considered in isolation since its behaviour is guaranteed to be the same regardless of the state of the rest of the program.

To demonstrate by way of example, consider the Java method given in figure 1.1 which sums the elements of an array of integers. Note its use of a **for** loop and the use of assignment statements, on lines 2 and 4, which change the value of a variable. In contrast, consider the Haskell function given in figure 1.2, which does not use statements or destructive updates, but rather relies on recursion and pattern-matching. On line 1 we match the case that the input list is empty. On line 2 we match the case that the input list is not empty, but rather has a head x (an integer) and a tail xs (a list of integers), with the resulting value being declared as an arithmetic expression involving those two. Finally, consider figure 1.3 which is the same function written in Pola. It is more verbose than that of the Haskell function and requires the "on-the-fly" introduction of a recursive function, f, to accomplish the task, but is structurally similar to that of the Haskell function. Line 2 matches the case where the list is empty (the Nil case) and line 3 matches the case where it is not empty (the Cons case), where it has a head z and a tail zs.

Practical general-purpose functional languages, such as Lisp, Haskell or Concurrent

```
1 | sumArray = x |  .fold f(y) as {
2 |     Nil.0;
3 |     Cons(z, - | zs).add(z, f(zs)) }
4 | in f(x);
```

Figure 1.3: A Pola function which sums the elements of a list of integers and returns that sum.

Clean, find some way to allow I/O—which is a side-effect—to be integrated into the functional framework of the language. This is currently absent in Pola, primarily as it would needlessly complicate the language and distract from the main focus of this thesis.

1.2 Implementation

An implementation of the Pola language can be downloaded and examined, available under a free software licence. The implementation is an interpreter written in the Haskell programming language. It performs parsing and type inference according to the specifications for the Pola language given in chapter 3 and evaluates expressions according to the operational semantics of chapter 3. Note that, in chapter 3, two different notations are given for the language: Compositional Pola, which is useful for exposing the rules of inference for the language with great precision; and Pola, which is the language the implementation targets, and is more useful for programming use.

The implementation also offers automated size and bounds inference, as discussed in chapter 5.

The Dangerous Pola variant of Pola discussed in section 6.3.3 is implemented, as it offers a greater level of expressiveness to the programmer than Pola. However, none of the other variants discussed in chapter 6 are supported by the implementation.

To aid the reader in navigating the source code, Pola's implementation has been written in a literate style, a style of programming developed by Donald Knuth to aid in programmer comprehension [24].

The reader is invited to explore the source code for the reference implementation of Pola at the following website and git code repository:

`https://gitlab.com/professormike/pola`

1.3 Summary

This thesis introduces a functional language, Pola, which is restricted to polynomial-time functions. Chapter 2 gives an overview of past research which this thesis builds upon, and previous ideas in a similar vein. Chapter 3 introduces Compositional Pola, a compositional language based on lambda calculus to provide the semantics and typing. Pola is introduced as syntactic sugar atop Compositional Pola. Novel contributions in this chapter are the type inference algorithm and proof of well-typing and the representation using compositional notation. The syntax, semantics and core typing involved collaboration with Robin Cockett and Brian Redmond, then both at the University of Calgary. Chapter 4 gives an overview of an implementation of Pola and references to download and examine the source code in greater detail. Chapter 5 gives a description of size and time bounds and gives an algorithm to automatically infer practical bounds. Proof of correctness and, as a corollary, a proof that the language is constrained to polynomial time is provided. The entirety of the chapter is a novel contribution. Chapter 6 discusses the limitations on expressiveness due to the type restrictions in the language and also

discusses variants of the language with different expressive and computational power. Chapter 7 gives a conclusion of the results and main contributions.

Chapter 2

Literature survey

Static analysis is the process of inferring, from a description of a computation, significant properties of that computation. The proof of the undecidability of the halting problem—i.e., the problem of giving an algorithm to universally decide if a Turing machine halts or not—provides an early upper bound on the capabilities of static analysis. An algorithm to determine how long any C program takes to run, for example, is clearly impossible, as the halting problem can be reduced to it.

Even if determining a running time were possible, it may not be useful. A static analysis which takes as long to compute as the computation under analysis is not of great use.

In spite of these barriers, there is a pressing need for static analysis. Software publishers might like to know how long a computation will take, or how much memory it will require, or, conversely, what the minimum hardware specifications are for some task. In embedded systems, for example, or systems with real-time requirements, it is imperative that it can be guaranteed with certainty that software will not exceed pre-determined requirements.

This chapter will deal with previous work in statically determining the consumption of resources, namely time and memory usage. These two resources are not the only resources that a piece of software might require, but they are the most general, and in any case the techniques used to determine them can be modified to determine other resource requirements.

2.1 Abstract interpretation

Abstract interpretation involves interpreting a language, usually a universally powerful language, in an "abstract" manner. For instance, consider the following C program:

```
static int foo(int x) {
    return x * 2;
}

int main(void) {
    int a = foo(3);
```

```
    int b = foo(4);
    return a + b;
}
```

We begin interpreting (or executing) the program from *main* as usual. One way to carry out abstract interpretation is to have a global binding for *foo*'s x variable, not dependent on any context. During the first function call to *foo*, x has a value of 3 and we assign a value of 6 to a. During the second function call to *foo*, x has a value of $3, 4$ (either 3 or 4, since we can't distinguish x based on where it's called from) and b gets a value of $6, 8$. We say, abstractly, that the program returns a value of $12, 14$.

Different abstract interpreters offer different abstract semantics. In abstract interpretation we "interpret" the program according to the usual program flow, i.e., for a C program we start "interpreting" at the first statement and move towards the last statement. Also, for any construct which offers the possibility of undecidability, we "abstract" the semantics to deal with sets of values instead of concrete values.

Gustafsson et al. provide a simple abstract semantics for C as described above, as a means of determining flow analysis [15]. The information from the flow analysis, determining the value ranges of variables before and after loops, is then used to help determine worst-case execution time.

The idea of abstract interpretation came from Cousot and Cousot [8] and had very strict semantics for a program represented as a lattice. It encompassed data flow analysis, in that all data flow analysis could be written as abstract interpretation [8]. This model has limited utility with functional languages due to the referential transparency provided by those languages.

2.2 Restricted models of computation

The implications of the undecidability of the halting problem are that, to determine the running time of a computation more generally, one must abandon the hope of certainty, or one must abandon the hope of universality. The latter is promising because much of the software written today is, in the grand scheme of computability theory, not very complex: many software problems would be in the class of polynomial time or maybe even more restricted complexity classes such as those in logarithmic space. Many of the algorithms in use today halt in a time polynomial with respect to their input, and so for many uses it will suffice to have a static analysis which only works on that class of algorithms.

Strategies for restricting computational power, while still allowing useful programs to be written, include limiting the growth of the functions, limiting the structure of the programs and the primitives provided and using types.

2.2.1 Primitive recursion

The class of primitive recursion functions is well-studied [31]. A function, f, with tuples of the natural numbers as domain and natural numbers as co-domain is primitive recursive if and only if one of the following is true:

1. $f(x) = 0$ (zero);
2. $f(x) = x + 1$ (successor);
3. $f(x_1, \ldots, x_n) = x_i$ for some $i, n \in \mathbb{N}$ such that $i \leq n$ (projection);
4. $f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \cdots, g_m(x_1, \ldots, x_n))$ for some $m, n \in \mathbb{N}$ where h and each g_i are primitive recursive (composition); or
5. $f(0, x_1, \ldots, x_n) = g(x_1, \ldots, x_n)$ and $f(y + 1, x_1, \ldots, x_n) = h(x_1, \ldots, x_n, y, f(y, x_1, \ldots, x_n))$ for some $n \in \mathbb{N}$ where g and h are primitive recursive functions (primitive recursion).

This general model of primitive recursion, of recursing over the structure of a datum (in this case, the structure of a natural number expressed in Peano arithmetic), will become a strong part of Pola, as seen in section 3.1.2.

Limited recursion and the Grzegorczyk hierarchy

A limited recursive function is a primitive recursive function, as described above, with the restriction that its value must be bounded by some base function, i.e., one that is not built up through recursion. We take the definition of primitive recursion above and modify clause 5. Consequently, the clauses for defining a function f are as follows:

1. $f(x) = 0$ (zero);
2. $f(x) = x + 1$ (successor);
3. $f(x_1, \ldots, x_n) = x_i$ for some $i, n \in \mathbb{N}$ such that $i \leq n$ (projection);
4. $f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \cdots, g_m(x_1, \ldots, x_n))$ for some $m, n \in \mathbb{N}$ where h and each g_i are primitive recursive (composition); or
5. $f(0, x_1, \ldots, x_n) = g(x_1, \ldots, x_n)$ and $f(y + 1, x_1, \ldots, x_n) = h(x_1, \ldots, x_n, y, f(y, x_1, \ldots, x_n))$ for some $n \in \mathbb{N}$ where g and h are primitive recursive functions and also $f(y, x_1, \ldots, x_n) \leq j(y, x_1, \ldots, x_n)$ for all $y, x_1, \ldots, x_n$ where j is some function previously defined by rules 1 to 4 (limited recursion).

The definition of limited recursion has great influence in the definition of the Grzegorczyk hierarchy, and has the unfortunate side-effect that it becomes undecidable whether or not some function is limited recursive.

The Grzegorczyk hierarchy [31] is a hierarchy of sets of functions. We first define a set of functions, E_i, for all $i \in \mathbb{N}$. E_0, the first function, has two parameters, $x, y \in \mathbb{N}$. All other E_i functions have a single parameter. The E_i functions are defined as follows:

$$E_0(x, y) = x + y \tag{2.1}$$
$$E_1(x) = x^2 + 2 \tag{2.2}$$
$$E_n(x) = E_{n-1}^x(2) \tag{2.3}$$

From these functions we can define the hierarchy of sets of functions, denoted $\mathcal{E}^i$ for some natural number i. The first set in the hierarchy, $\mathcal{E}^0$, is equal to the zero functions, the successor functions, the projection functions, the composition functions (over functions in $\mathcal{E}^0$), and the limited recursive functions over functions in $\mathcal{E}^0$. Of note, no E_i functions are in $\mathcal{E}^0$. Stated precisely, a function, f, is in $\mathcal{E}^0$ if and only if:

1. $f(x) = 0$ (zero);

2. $f(x) = x + 1$ (successor);

3. $f(x_1, \ldots, x_m) = x_i$ for some $i, n \in \mathbb{N}$ such that $i \leq n$ (projection);

4. $f(x_1, \ldots, x_m) = h(g_1(x_1, \ldots, x_m), \cdots, g_p(x_1, \ldots, x_m))$ for some $m, p \in \mathbb{N}$ where h and each g_i are in $\mathcal{E}^0$ (composition); or

5. $f(0, x_1, \ldots, x_m) = g(x_1, \ldots, x_m)$ and
   $f(y+1, x_1, \ldots, x_m) = h(x_1, \ldots, x_m, y, f(y, x_1, \ldots, x_m))$ for some $n \in \mathbb{N}$ where g and h are in $\mathcal{E}^0$ and also $f(y, x_1, \ldots, x_m) \leq j(y, x_1, \ldots, x_m)$ for all $y, x_1, \ldots, x_m$ where j is in $\mathcal{E}^0$ (limited recursion).

For each $n > 0$, we define $\mathcal{E}^n$ as the basic functions (zero, successor, projection), composition functions (over functions in $\mathcal{E}^n$), limited recursive functions (ibid), as well as the functions E_0 and E_n. Stated precisely, a function, f, is in $\mathcal{E}^n$ for some $n \in \mathbb{N}, n > 0$ if and only if:

1. $f(x) = 0$ (zero);

2. $f(x) = x + 1$ (successor);

3. $f(x_1, \ldots, x_n) = x_i$ for some $i, n \in \mathbb{N}$ such that $i \leq n$ (projection);

4. $f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \cdots, g_m(x_1, \ldots, x_n))$ for some $m, n \in \mathbb{N}$ where h and each g_i are in $\mathcal{E}^n$ (composition); or

5. $f(0, x_1, \ldots, x_n) = g(x_1, \ldots, x_n)$ and
   $f(y + 1, x_1, \ldots, x_n) = h(x_1, \ldots, x_n, y, f(y, x_1, \ldots, x_n))$ for some $n \in \mathbb{N}$ where g and h are in $\mathcal{E}^n$ and also $f(y, x_1, \ldots, x_n) \leq j(y, x_1, \ldots, x_n)$ for all $y, x_1, \ldots, x_n$ where j is in $\mathcal{E}^n$ (limited recursion); or

6. $f = E_k$ for $k < n$.

Note that $\mathcal{E}^1$ includes all functions in which the result is a constant number of additions of its parameters, $\mathcal{E}^2$ includes all functions in which the result is a constant number of multiplications or additions of its parameters, $\mathcal{E}^3$ includes all functions in which the result is a constant number of exponentiations, multiplications or additions, and so on.

Predicting time complexity of recursive functions

Restricted models of computation can prove useful for predicting time complexity of functions. The Grzegorczyk hierarchy itself comprises the exponential hierarchy [2], though there is the possibility to provide more precise predictions.

R.W. Ritchie [30] was one of the first to formalize predicting resource requirements for functions in a primitive recursive scheme. He created a hierarchy $F_1, F_2, F_3, \ldots$ between $\mathcal{E}^2$ and $\mathcal{E}^3$. Where $F_0 = \mathcal{E}^2$, a function $f \in F_1$ if $\exists g \in F_0$ such that, if $T_f(x)$ is the time required to compute f, then $T_f(x) \leq g(x)$. Similarly, for all $i \in \mathbb{N}$, $f \in F_{i+1}$ if $\exists g \in F_i$ such that $T_f(x) \leq g(x)$ for all x.

Perhaps more interesting, however, was Ritchie's formalizing of a method to express the running time of primitive recursive functions, using a very abstract computational model for determining "time." For the base functions, the definitions of T_f are more obvious. For composition and primitive recursion, though, they are less obvious. He showed that:

- if $f(x, y) = h(x, g(y))$, then $T_f(x, y) = |x| + |y| + \max\{|x| + T_g(y), T_h(x, g(y))\}$; and
- if $f(0, x) = g(x); f(y + 1, x) = h(x, y, f(y, x))$, then $T_f(y, x) = 2(|x| + |y| + \max\{\max_{z \leq y}\{T_h(x, z, f(z, x))\}, T_g(x)\}$.

Limitations to primitive recursion

Loïc Colson showed some practical concerns with using primitive recursion as a basis for programming languages [7]. The function:

$$\min(a, b) = \begin{cases} a & , \text{ if } a < b \\ b & , \text{ otherwise} \end{cases}$$

can be implemented quite easily in a primitive recursive framework. However, there does not exist a primitive recursion algorithm which will compute it in $O(\min(a, b))$ time. While this specific instance can be resolved easily in a practical programming language, for example by the introduction of a constant-time less-than ('<') operator, it neglects the wider problem presented, namely that in a primitive recursive scheme, one cannot efficiently recurse through two data simultaneously. Any function which needs to do something of the sort has a greater time complexity in primitive recursive frameworks than in unrestricted computational frameworks [7].

2.2.2 Hofmann survey

Martin Hofmann presents an excellent survey of programming languages capturing complexity classes [21]. A programming language is said to capture a complexity class $\mathcal{C}$ if every well-defined program in that language has time complexity in $\mathcal{C}$. Of especial note is that in the introduction to this survey, Hofmann specifically brings up the example of guaranteeing resource restrictions in embedded systems.

He starts off with a paper by Alan Cobham [4]. This is a seminal paper on restricting computational power to limit programs to what he felt was intrinsic feasibility. Cobham used a scheme of limited recursion, except that it recursed "on notation," i.e., it

recursed on the length of x (when written as a string), not on the value of x. Successor functions were notation-wise, so if the number were represented in binary, there would be two successor functions, $S_0(x) = 2x$ and $S_1(x) = 2x + 1$, effectively adding a '0' or '1,' respectively, to the end of x. Cobham showed that these successor functions, zero functions, composition functions, projection functions, limited recursion by notation, and the smash function, $x\#y = 2^{|x|\cdot|y|}$, equal the polynomial-time functions.

The same paper also showed that all functions in $\mathcal{E}^2$, the third set in the Grzegorczyk hierarchy, require at most linear space.

Gurevich showed that primitive recursion over a finite domain, i.e., where the successor function has an upper limit, is exactly the class of programs operating in logarithmic space [14].

The two previous models can be regarded as being somewhat awkward or artificially restricted, Cobham's method especially because it is undecidable whether a program is well-formed or not. Bellantoni and Cook provide a model based on primitive recursion more naturally bounded to polynomial-time functions [1]. Function parameters are divided into "normal" and "safe" parameters, denoted as $f(x; y)$, where parameters before the semi-colon (x) are "normal" and parameters after the semi-colon (y) are "safe." Normal parameters may recursed over, but safe parameters may only be used in composition. Further, the result of a recursive function may only be used in composition.

While Pola does not directly use the idea of "normal" and "safe" parameters, it does borrow the notion of having two classes of variables which have differing restrictions on them, as will be seen in section 3.1.2.

Loop programs

One model of computation proposed in the late 1960s is that of loop programs, which is equivalent in power to the class of the primitive recursive functions [26]. A loop program operates over an aribtrary, but finite, number of named registers (canonically X and Y and so on). Each register holds a natural number of unbounded size. A loop program is a finite sequence of instructions, where each instruction is one of:

1. $X = Y$ for two registers X and Y;
2. $X = X + 1$ for some register X;
3. $X = 0$ for some register X;
4. $LOOP\ X$ for some register X; or
5. END.

There is a further restriction upon the structure of loop programs, which is that $LOOP\,X$ and END instructions must be paired.

The semantics of loop programs are as can be expected. $X = Y$ instructions overwrite the contents of register X with the contents of register Y. The $X = X + 1$ increments by one the contents of register X. $X = 0$ instructions set the value of register X to zero. For a block of instructions, $LOOP\ X, I_1, \ldots, I_n, END$ where the $LOOP\ X$ and END

instructions are "paired," the instructions $I_1, \ldots, I_n$ will be executed serially X times. For instance, if X has the value 3 when such a block is encountered, the instructions $I_1, \ldots, I_n, I_1, \ldots, I_n, I_1 \ldots, I_n$ will be executed. The variable which is used to control the loop (X, in this case) must be invariant.

2.2.3 The Hume languages

Hume, the Higher-order Unified Meta-Environment, is a hierarchy of programming languages aimed at writing correct and verifiable code for embedded systems. The simplest layer of Hume, HW-Hume, disallows recursion entirely, allowing only constant transformations between input bits and output bits. FSM-Hume allows non-recursive first-order functions and non-recursive data types. Template-Hume allows pre-defined higher-order functions (such as *map* or *iter*, found in typical functional languages), user-defined first-order functions, and inductive data types. PR-Hume allows primitive recursive functions over inductive data types. Finally, Full Hume is an unrestricted, Turing-complete language [17].

Michaelson early on develops a method for bounding recursion in a simple lambda-calculus–based language, either by eliminating recursion entirely or by allowing a primitive recursion [27].

Hardware level

The lowest level of Hume is HW-Hume (hardware Hume), which provides simple pattern bindings [16]. For instance, a parity checker can be written as follows:

```
box parity
in (b1 :: Bit, b2 :: Bit)
out (p :: Bit)
match
  (0, 0) -> 0
| (0, 1) -> 1
| (1, 0) -> 1
| (1, 1) -> 0
```

No recursion or iteration is offered [16], which limits its computational utility.

Finite state machine

The finite state machine level of Hume, FSM-Hume, provides a syntax for describing the behaviour of generalized finite state machines (abstracted to inductive data types in general, not just symbols) in the expression syntax of Hume [18]. Figure 2.2.3 contains a finite state machine for determining if a binary number is divisible by 3.

In FSM-Hume, this would be written as:

```
data State = Rem0 | Rem1 | Rem2;
type Bit = Word 1;
box div3
```

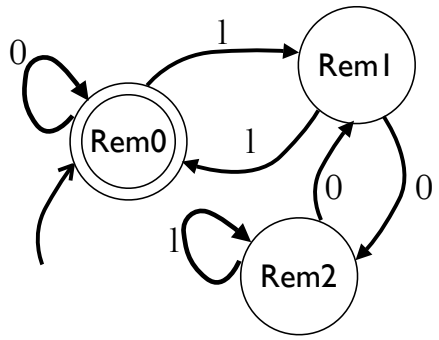

Figure 2.1: A finite state machine to decide if a binary number is divisible by 3.

```
in (s :: State, inp :: Bit)
out (s' :: State, out :: Bool)
match
  (Rem0, 0) -> (Rem0, True)
| (Rem0, 1) -> (Rem1, False)
| (Rem1, 0) -> (Rem2, False)
| (Rem1, 1) -> (Rem0, True)
| (Rem2, 0) -> (Rem1, False)
| (Rem2, 1) -> (Rem2, False)
```

The remainder of the program must happen at one level lower, in HW-Hume, to wire the new "box."

```
stream input from "std_in";
stream output to "std_out";
wire input to div3.inp;
wire div3.s' to div3.s initially Rem0;
wire div3.out to output;
```

Templates

Applying a finite transducer to input, as in FSM-Hume, works for simple examples, but becomes tedious when trying to apply the same transformation to each bit in a vector, for instance. Template-Hume offers some pre-defined higher-order functions to achieve this end [18].

For instance, to square each number in a vector:

```
square :: Int -> Int;
square x = x * x;
squareVector :: vector m of Int -> vector m of Int;
squareVector v = mapvec square v;
```

This makes use of the predefined higher-order function *mapvec*.

Primitive recursion

PR-Hume is a typical functional programming language, permitting recursion, but bounding the recursion to the class of primitive recursive functions [18]. Since determining if an arbitrary function is primitive recursive is undecidable and PR-Hume offers no structural restrictions to bound the recursion, PR-Hume relies on partial analysis to determine if a function is primitive recursive [18].

The mechanism for turning unrestricted expressions into forms that guarantee primitive recursion relies on single-step reductions and looking for syntactic expressions of the form:

```
pr y x = f y x (pr (h y) (g x))
```

where y is the variable which must strictly decrease in size, h is a "size-reducing" function and f and g are functions already analysed to be primitive recursive [27].

2.2.4 Categorical approaches

Category theory has offered a few approaches to creating restricted programming languages, often called categorical computing. Common among categorical approaches is the use of inductive and coinductive datatypes.

For instance, stateless attribute grammars used in parsing can be written as catamorphisms [12]. The grammars are written as inductive data types, with each alternative in a grammar rule corresponding to a constructor in an inductive datatype. Parsing then is taken as a catamorphism over this datatype [12].

Paramorphisms

One method of bounding recursion is to disallow explicit recursion entirely and allow primitives for performing recursion. As Dijkstra famously warned of the perils of goto statements [11], Meijer et al. warn of the perils of unrestricted recursion [25]. They provide a proposal to build more structured programs, based on category theory and unrestricted inductive data types, using various primitives for recursion.

Catamorphisms are a class of functions which replace the constructors of an inductively defined data structure, effectively recursing through each node in the structure. For instance, consider the inductive definition of a binary tree:

$$T(\alpha) := \mathit{Leaf} \mid \mathit{Node}(\alpha, T(\alpha), T(\alpha))$$

A catamorphism to sum the nodes of a tree could, in a typical language allowing recursion, be written as:

$$c(v) = \begin{cases} 0 & , \text{ if } v = \mathit{Leaf} \\ v_a + c(v_1) + c(v_2) & , \text{ if } v = \mathit{Node}(v_a, v_1, v_2) \end{cases}$$

For brevity and clarity, the authors suggest the use of "banana brackets" to write catamorphisms, so that the above function would be written as:

$$c = (\!|0, \lambda a.\lambda l.\lambda r.a + l + r|\!)$$

Note that the expression describing the *Leaf* component of the banana bracket is nullary (i.e., is abstracted over no parameters) whereas the expression describing the Node component of the banana bracket is ternary, corresponding to the fact that the Node constructor has 3 subterms.

Anamorphisms are a class of functions which allow one to construct a member of an inductive type from a seed value. In the context of a function $g : \alpha \to (\beta, \alpha, \alpha)$ and a predicate function $p : \alpha \to \mathsf{Bool}$, an example anamorphism might be written in a typical functional language as:

$$f(v) = \begin{cases} \mathit{Leaf} & , \text{ if } p(v) \\ \mathit{Node}(a, l, r) & , \text{ otherwise, where } (a, l, r) = g(v) \end{cases}$$

One thing to note is that even though anamorphisms often produce data structures of infinite size, in a lazy language, a catamorphism which acts on an anamorphism will always halt. Secondly, unlike with catamorphisms, anamorphisms are almost always given in the context of a function, in this case g.

The authors use "concave lenses" to denote anamorphisms, and the above example would be written as:

$$a = [\![g, p]\!]$$

A sort of combination of the two, holomorphisms are recursive functions in which the call tree is isomorphic to an inductive data structure. For instance:

$$h(v) = \begin{cases} c & , \text{ if } p(v) \\ v_a + h(v_1) + h(v_2) & , \text{ otherwise, where} (v_a, v_1, v_2) = g(v) \end{cases}$$

This is written using "envelope" notation, and the authors write it as:

$$h = [\![c, \lambda a.\lambda l.\lambda r.a + l + r), (g, p)]\!]$$

They also show that for any data type and for any a, b, c, d:

$$[\![(a, b), (c, d)]\!] = (\!|a, b|\!) \circ [\![c, d]\!]$$

Meijer et al.'s paper was the first to formally introduce paramorphisms to a programming language. A paramorphism is exactly like a catamorphism but the constituent functions accept extra arguments for attributes which have not undergone recursion. For instance, consider the factorial function over the unary natural numbers:

$$f(n) = \begin{cases} 1 & , \text{ if } n = \mathit{Zero} \\ (1 + p) \cdot f(p) & , \text{ if } n = \mathit{Succ}(p) \end{cases}$$

Note that the attribute p is used both on its own and as an argument to the recursion. Being able to use p on its own is what distinguishes this from a catamorphism. The authors propose "barbed wire" notation for writing paramorphisms, and this would be written as:

$$f = \{1, \lambda p \hat{p}.(1 + \hat{p}) \cdot p\}$$

These paramorphisms strongly relate to the fold constructs in Pola, as introduced in section 3.1.2.

Finally the authors provide operational semantics and prove termination properties and categorical properties of the recursive structures [25].

Charity

Charity is the most successful categorical programming language. It is centred around inductive and coinductive datatypes, with computations performed using catamorphisms ("folds") and anamorphisms ("unfolds") [5, 13]. It is not universally powerful—every program must halt given finite user input—but it is strictly more powerful than primitive recursion.

More properly stated, Charity offers three combinators (and their duals): the "fold" or catamorphism ("unfold" or anamorphism), map (also a "map" in the dual), and case (record). Maps and cases (records) are special cases of catamorphisms (anamorphisms), however.

A Master's student of Cockett's extended the language to Higher-order Charity [34]. Inductive types were left unchanged, but coinductive types could have their destructors parameterized over some higher-order function, which led to efficient *min* and *zip* functions. Further, Higher-order Charity offered an object-oriented system, built on top of higher-order coinductive types, where an unapplied anamorphism was a class, an applied anamorphism was an instance, and a parameterized destructor was a method.

Cockett describes the workings of the standard Charity interpreter, describing the basic datatypes (natural numbers, lists, trees) and co-datatypes (co-natural numbers, co-lists, co-trees). He also explains how to use common programming idioms in Charity such as "dualing" (converting between a datatype and its cognate co-datatype) and dynamic programming in Charity [5].

As a proof that Charity is more powerful than primitive recursion, a program to calculate Ackermann's function is provided [6]. Ackermann's function is defined as:

$$ack(0, n) = n + 1 \tag{2.4}$$
$$ack(m + 1, 0) = ack(m, 1) \tag{2.5}$$
$$ack(m + 1, n + 1) = ack(m, ack(m + 1, n)) \tag{2.6}$$

One can consider an infinite table, where the value at column m and row n is $ack(m, n)$. Equation 2.4 says that each entry has value $n + 1$ in column 0 for row n:

	0	⋯
0	1	⋯
1	2	…
⋮	⋮	⋱

In other words, column 0 is $tail(nats)$ where *nats* is the infinite list of natural numbers. Equation 2.5 says that each entry in row 0 has the same value as in row 1 of the previous column:

	0	1	⋯
0	1	2	⋯
1	2		⋯
2	3		⋯
⋮	⋮	⋮	⋱

Finally, equation 2.6 says that for some other row and column $m+1$ and $n+1$, we look in the table at co-ordinates $(m+1, n)$ to find value i, then look again in the table at co-ordinates (m, i) to find the new value at $(m+1, n+1)$. Since all values needed have been previously defined, this is a dynamic programming problem, and we end up with the correct infinite table of:

	0	1	2	3	$\cdots$
0	1	2	3	5	$\cdots$
1	2	3	5	13	$\cdots$
2	3	4	7	29	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Cockett's Ackerman construction creates a Charity function, `col_m`, which takes as its argument the infinite coinductive list of the previous column. It uses an unfold to destruct that list into a new infinite coinductive list. With the use of another function to retrieve an element from the infinite list at a particular index, it follows that we can compute $ack(m, n)$ for any given natural numbers m and n, thus making Charity able to compute functions beyond primitive recursive, while still remaining recursive.

Like Charity, Pola does have possibly-infinite coinductive structures much like Charity does, as will be seen in section 3.1.3, but will have typing restrictions to ensure that nothing super-polynomial can be produced through them.

2.3 Type theoretic approaches to determining resource consumption

A popular area of research in attempting to track memory usage, or resource consumption in general, is type theory. For instance, Herding's Master's thesis describes how to augment functions in the specification language CASL to track memory state [19]. The general idea is to encode resource consumption information into the type of a symbol (e.g., a variable or function). This allows composing components of a larger program together using type matching or type inference to gain information on the resource consumption of the program as a whole. We will see in chapter 5 that considering resource consumption to be a part of a symbol's type information will become useful for Pola.

2.3.1 Stack language

Saabas and Uustalu offer a type-theoretic approach to stack-based programs [33, 32], interesting because it deals specifically with programs with jumps ("gotos") without any explicit structure. The major motivation is to provide techniques for generating proof-carrying code for low-level programs.

The first paper dealt with simple unstructured code with gotos [33]. The second paper extended the language to allow operand stack operations [32].

The PUSH language described in [33] defines "a piece of code" (program) to be a finite subset of $\mathcal{P}(\mathbb{N} \times \mathbf{Instr})$, i.e., a set of pairs of labels and instructions. It generally deals with what it calls "well-defined pieces of code," where each label defines a unique

instruction. **Instr**, the set of instructions, is defined as the instructions load, store, push, goto (unconditional jump), gotoF (conditional upon the top operand, read as "goto if false"), plus some simple stack-based logical and arithmetic operations (add, eq, and so on). There is also a finite set of registers used by the load, store and push instructions. All of load, store and push have a single register operand; goto and gotoF have natural-number constants as their sole operands; add, eq and so on take their operands from the stack and do not deal with registers at all.

The authors structure the Push code into a finite union of non-overlapping labelled instructions, creating a new language called SPush, where an SPush program is a binary tree of labelled instructions, i.e., a binary tree where each leaf is a $(\mathbb{N}, \mathbf{Instr})$ pair. Construction is associative. The semantics are identical between Push and SPush, but using SPush allows the definition of a Hoare logic through composition. The authors construct a sound and complete Hoare logic.

Finally, the authors weaken the Hoare logic into a typing system. The type system ensures safe stack usage [33]. Any operations beyond stack usage are not considered.

2.3.2 Memory bounds

Hofmann and Jost provide a method for finding linear bounds on memory allocation in a first-order functional language, called LF, but in the context of explicit memory deallocation and in the presence of a garbage collector [22]. The authors excuse the restriction to first-order functions foremost because it is fruitful to do so, but also because one can work around it through code duplication. They introduce types with simple first-order functions in which data must be units (the data type with only one datum in its domain, often written as 1 or () in functional languages), Booleans, lists, Cartesian products, or unions:

$$A := 1 \mid \mathsf{B} \mid \mathsf{L}(A) \mid A \otimes A \mid A + A \tag{2.7}$$

$$F := (A, \ldots, A) \to A \tag{2.8}$$

where A is a zero-order type and F is a first-order type. The function SIZE$(-)$, which returns the natural number indicating the number of memory cells needed to store a datum of that type, is defined on all zero-order types.

A defining characteristic of the LF language presented is that it has two methods for pattern-matching: the match construct (destructive pattern matching) and the match' construct (non-destructive pattern matching) [22]. In the former, the datum being matched is deallocated from memory, while in the latter, the datum being matched can be referenced in subsequent parts of the program.

For the purposes of determining memory usage, the authors define a new language, $\mathsf{LF}_\Diamond$, in which the data types are augmented with size annotations, where $\mathbb{Q}^+$ is the set of positive rational numbers:

$$A := 1 \mid \mathsf{B} \mid A \otimes A \mid R + R \mid \mathsf{L}(R) \tag{2.9}$$

$$R := (A, k) \text{ for some } k \in \mathbb{Q}^+ \tag{2.10}$$

$$F := (A, \ldots, A, k) \to R \text{ for some } k \in \mathbb{Q}^+ \tag{2.11}$$

The R types are read as "rich types" because they carry information about their memory requirements. Note that functions accept parameters of "plain" (A) types, but return a rich value. They set up type judgements to determine resource requirements. For instance, the simple rule which types the expression () to be of the unit type:

$$\frac{n \geq n'}{\Gamma, n \vdash_\Sigma () : 1, n'}$$

is read as "in the context Σ under environment Γ with n free memory cells, () is of type 1 with n' or more unused memory cells left over." More interesting cases are to see how memory requirements change when doing a destructive pattern match on a list (equation 2.12) as compared to a non-destructive pattern match on a list (equation 2.13):

$$\frac{\Gamma, n \vdash_\Sigma e_1 : C, n' \quad \Gamma, x_h : A, x_t : \mathsf{L}(A,k), n + \mathrm{SIZE}(A \otimes \mathsf{L}(A,k)) + k \vdash_\Sigma e_2 : C, n'}{\Gamma, x : \mathsf{L}(A,k), n \vdash_\Sigma \mathsf{match}\ x\ \mathsf{with}\ (nil \Rightarrow e_1; cons(x_h, x_t) \Rightarrow e_2) : C, n'} \quad (2.12)$$

$$\frac{\Gamma, n \vdash_\Sigma e_1 : C, n' \quad \Gamma, x_h : A, x_t : \mathsf{L}(A,k), n + k \vdash_\Sigma e_2 : C, n'}{\Gamma, x : \mathsf{L}(A,k), n \vdash_\Sigma \mathsf{match}\ x\ \mathsf{with}\ (nil \Rightarrow e_1; cons(x_h, x_t) \Rightarrow e_2) : C, n'} \quad (2.13)$$

Finally, the authors devise a system for inferring annotations in the program, such that the programmer does not necessarily need to explicitly annotate sizes for each function or datum.

Hofmann and Jost also introduce an amortized system of determining memory bounds [23]. They propose a Java-like language where each value carries with it some "potential" and each operation carries with it some amortized costs. The sum of the amortized costs plus the potential of the input to an operation then provides a bound.

Each class is augmented into a "refined class," which is a class that has some potentials attached, through what are called views. For instance, in the view a, the *Cons* class carries a potential of 1 (indicating that operations on it have an extra cost of 1, stripping off the potential associated with the *Cons*) whereas in another view b, the same *Cons* class carries a potential of 0. Note that a single object in the Java-like language can be, and often is, considered from different views with different potentials and thus can be of different types.

A view carries three pieces of information: the potential of the class itself, the potential of each attribute (where each attribute has both a "get-view," the potential of reading the attribute, and a "set-view," the potential of writing to the attribute) and the potentials for each of its methods, which are functions of its arguments.

For instance, consider a program to append two lists:

```
abstract class List { abstract List append(List x); };
class Nil { List append(List x) { return x; } };
class Cons {
  int head;
  List tail;
  List append(List x) {
    List r = new Cons();
```

```
    r.head = this.head;
    r.tail = this.tail.append(x);
    return r;
  }
};
```

We introduce two views: a and b. In the view a, Cons has potential 1 and all other classes have potential 0; in the view b, all classes have potential 0. The method append is well-defined only as $\mathsf{List}^a \times \mathsf{List}^b \to \mathsf{List}^b$, i.e., it is only defined as operating on lists in the view a, taking arguments in the view b and returning values in the view b.

To prove (correctly) that summing a list consumes memory proportional to its first operand, but irrespective of its second operand, we need to prove judgements such as:

$$this : \mathsf{Nil}^a, x : \mathsf{List}^b \vdash_0^0 (\text{return } x) : \mathsf{List}^b \tag{2.14}$$

$$this : \mathsf{Cons}^a, x : \mathsf{List}^b \vdash_0^1 (\mathsf{List}\ r = \text{new...}) : \mathsf{List}^b \tag{2.15}$$

Here the notation $\vdash_{m'}^{m}$ carries the semantics that the amount of free heap space must be at least m plus the potential of the expression given on the right-hand side, and that the m plus the potential of the result is at most the amount of free heap space after execution. In this case, since the operand is in the a view, it carries the cost of the execution in terms of its potential (e.g., since the potential of Cons is 1, a list of length 5 will have 5 Conses and thus have a potential of 5) and the cost of the *append* function will be 1 for each potential of the input.

The authors prove soundness of the typing system. One nice feature of the system is that it is not weak with respect to divide-and-conquer methods as similar systems are. For instance, when writing Quicksort in the naïve way (choosing the first element of a list as the pivot), one often has to judge that, after splitting up the remainder of the list, both lists have maximum size $n - 1$, where n is the length of the list. When merging the two lists back together, one finds a bound on the size of the list as $2n - 1$, rather than the proper n. This can erroneously lead one to believe that Quicksort consumes exponential space. In this system, one requires explicit judgements for contraction, so that when aliasing objects, the potential is divided among them [23].

The major downside of the Hofmann and Jost system is that memory bound inference is not considered and neither is any type inference system.

2.4 Static memory management

Particularly as memory accesses become increasingly costly in future computer architectures, alternative memory management models are being explored. Memory management has moved from simple stack-based models [10] to explicit heap management via C's `malloc` and `free` to trace-based garbage collection schemes used in most popular languages today such as Java, C♯, Lisp or Haskell. However, in the interest of increased performance, namely fewer memory accesses, and better safety, there is a push to make memory management more "static" once again.

Region-based memory management is a memory management model added to the ML language [36]. It annotates ML types and ML expressions with region variables where a region variable refers to a statically allocated and deallocated region of memory. Unlike in a simple stack-based memory management system, we have a finite stack of not just finite memory cells, but a finite stack of unbounded *regions* of memory which can grow dynamically. Like a simple stack-based memory management system, the regions are lexically scoped and deallocated in constant time.

The major changes to the language are the `at` expression and the `letregion` expression. The `letregion` expression creates a new region, and introduces a corresponding region variable, which exists only within the lexical scoping of the `letregion` expression. The `at` expression is used for atomic constructions indicating which region to allocate it in. As an example:

$$(\lambda\, x.\mathit{letregion}\ \rho\ \mathit{in}\ \lambda y.(y, \mathsf{Nil}@\rho)\ \mathit{at}\ \rho\ \mathit{end}\ x)\ 10$$

This function creates a region, ρ, and allocates an object, Nil, in that region. Once the expression is evaluated, ρ is deallocated (and the Nil is destroyed with it).

A fully annotated region-based expression becomes quite unwieldy and there has been work done to automatically infer which regions to place objects in [35].

2.5 Conclusion

There has been work done in the past to study both static analysis techniques on existing programming languages and in developing more explicitly or rigidly structured programs to aid in resource analysis. Developing a flexible programming model that lends itself to automatic inference of resource requirements is still something missing in today's programming languages, however.

The next chapter will now introduce Pola, a programming language which borrows some aspects from past work, such as restricted recursion and typing restrictions on different classes of variables, and adds novel ideas to provide a practical programming language restricted to polynomial time. This will allow future chapters to provide an method to infer running-time bounds and memory-consumption bounds.

Chapter 3

Compositional Pola

Twe expositions of a functional language constrained to polynomial-time functions are presented here, Pola and Compositional Pola. Both languages are equivalent in their typing and semantics. Pola is the variant which has been implemented (see chapter 4) and would be used by programmers. However, the language constructs contained within it would be far more complicated to describe and so, in the interest of clarity, a compositional form of the language called Compositional Pola is described in this chapter, which simplifies and clarifies exposition, definitions and proofs of the language. For most of the chapter, and for all of the introductory examples, we will present examples using the syntaxes of both Pola and Compositional Pola. Pola can be thought of as a "sugared" Compositional Pola, a syntax which makes the constructs more readable. Proofs and detailed descriptions will be based off of Compositional Pola only, however.

3.1 Introductory examples

3.1.1 Recursion-free examples

Figure 3.1 shows simple Pola code that does not use any recursion or complicated data types. It attempts to define two different functions, one called f and the other called g. Both functions aim to have a single parameter, called x, and return a tuple that contains the value of x in both positions of the tuple. The syntax for defining functions in Pola is to begin with the name of the function, followed by an equal sign, following by a list of parameters segregated such that opponent parameters are on the left of a vertical bar and player parameters on the right, followed by a dot, followed by a term.

The example demonstrates one crucial part of Pola which permeates the language: two different types of variables. The function f in figure 3.1 declares the parameter x

$$\begin{array}{r|l} 1 & f = x \mid \ .(x, x); \\ 2 & g = \ \mid x.(x, x); \end{array}$$

Figure 3.1: A demonstration of the distinction between opponent and player variables. The function f is legal in Pola, but the function g is illegal.

```
1  | data Nat → c
2  |     = Zero :→ c
3  |     | Succ : c → c;
4  | data List(a) → c
5  |     = Nil :→ c
6  |     | Cons : a, c → c;
7  | add = x | y.fold f(a, b) as {
8  |     Zero.b;
9  |     Succ(− | n).f(n, Succ(b)) }
10 | in f(x, y);
11 | append = l | m.fold f(a, z) as {
12 |     Nil.z;
13 |     Cons(x1, − | x2).Cons(x1, f(x2, z)) }
14 | in f(l, m);
```

Figure 3.2: An example of inductive data types and folds in Pola.

before the vertical bar, making it an *opponent* variable. An opponent variable has no restrictions on where it can be used or how often it can be used. Function f is therefore a legal function.

The function g, in contrast, declares its parameter x after the vertical bar, designating it as a *player* variable. Player variables have an important restriction on their use, namely that they cannot be repeated multiple times in the same term. This will later prove to be an invaluable tool in restricting the computational power of the language. Because the function g attempts to use the parameter x twice, the function g is illegal in Pola.

3.1.2 Inductive data types and folds

Figure 3.2 gives an example of a Pola program which uses inductive data types and folds, inductive data types being those which can be constructed inductively (e.g., Booleans, characters, integers, strings, lists, trees) and folds being the mechanism for recursion over inductive data structures. The program in figure 3.2 defines two inductive data types, those of the natural numbers and lists, and defines two functions, the *add* function which adds two natural numbers, and the *append* function which appends two lists together. At this point it is worthy of mention that Pola does not have any predefined primitive types (such as integers) nor any predefined primitive operators (such as addition). Though the language could easily be extended to offer such things in the interest of efficiency, that is not considered in this thesis because it distracts from the main contribution of the thesis.

After lines 1–3, the type Nat will be defined. The expression Zero is of type Nat; the expression Succ(e) will be of type Nat for any expression e which is also of type Nat. Within the definition of the type itself, the type variable c is used to stand recursively for that of the type being defined: in this case, the type variable c stands for the type Nat. Consequently, Zero : $\rightarrow c$ can be read as defining the constructor Zero, which takes no parameters (there are no types to the left of the $\rightarrow$ symbol) and produces a value

of type Nat. Succ : $c \to c$ can be read as defining a constructor, Succ, which takes one parameter of type Nat and produces a value of type Nat.

From these two constructors one can define the natural numbers. From this point forward we will, for clarity, denote natural numbers constructed from these expressions as Arabic numerals with an overhead bar. For instance, the expression Succ(Succ(Zero)) will be denoted as $\bar{2}$.

Lines 4–6 define the inductive type of lists. The expression Nil is of type List(a) for all types a. The expression Cons(h, t) is of type List(a) for all expressions h of type a and expressions t of type List(a). For example, the expression Cons(Zero, Nil) is of type List(Nat). a in this context is a type variable—not itself a concrete type—and List(a) is a parameterized type.

From this point forward we will, for clarity, denote lists constructed from these expressions within square brackets. For instance, the expression Cons($\bar{3}$, Cons($\bar{7}$, Nil)) will be denoted as $[\bar{3}, \bar{7}]$.

Lines 7–10 define the function *add*. On line 7, x and y are introduced as parameters to the function. x being placed to the left of the vertical bar (|) denotes it as an "opponent" variable and y being placed to the right of the vertical bar denotes it as a "player" variable. Opponent variables are used to drive recursion, and consequently x is required to be an opponent variable in this case. The remaining parameter, y, could be declared either an opponent or player variable and the addition function would work properly in either case, though it is often preferred to have it as a player variable, for reasons that will be made clear in section 3.4. Since y is a player variable in this function, it cannot drive a recursion, which is acceptable in this case as there is no need for y to drive a recursion.

The **fold** keyword allows the introduction of an "on-the-fly" recursive function, f. Within the scope of the **fold** construct we may refer to f to recurse, under some typing restrictions that will be explained in section 3.1.2 and section 3.4. a and b are the parameters to the recursive function, f, with a (the first parameter named) being the variable which is being recursed over, and b (and any number of other parameters) being extra parameters to the recursive function. Every parameter of a **fold** function is a player parameter. Within the **fold** is an implicit pattern-matching on the first parameter, a. If a is of value Zero then we will evaluate the expression on line 8; otherwise we will evaluate the expression on line 9. The "coda" to the **fold** construct, given on line 10, provides initial values to the function f.

It should be noted at this point that, in Pola, it is an error to refer to a function which has not yet been defined. For instance, it would be a typing error for the function *add* to be referred to within the definition of *add* itself. It would also be an error for the function *add* to refer to the function *append*, as that function would not be defined at that point. Referencing previously defined functions is allowed by the typing system, however: *append* could make reference to the function *add*.

The pattern-matching syntax on line 9 requires explanation. Succ($- \mid n$) should be read as matching if the first parameter, a, is a Succ value. The parameter to the Succ constructor is bound twice: once in the opponent context and once in the player context. In this case, which is the usual case, the variable is not of use when bound in the opponent context, and so we bind it to the anonymous variable, $-$. Variable n, then, is bound in

```
1 | add = x | y.fold f(a, b) as {
2 |     Zero.b;
3 |     Succ(− | n).f(n, Succ(b)) }
4 | in f(x, y);
5 | mul = x, y |  .fold f(a, b) as {
6 |     Zero.b;
7 |     Succ(− | n).f(n, add(y, b)) }
8 | in f(x, Zero);
```

Figure 3.3: Addition and multiplication functions defined in Pola.

the player context. To demonstrate by example, suppose that the initial values to the function *add* were $x = \bar{3}$ and $y = \bar{1}$. The initial values within the recursive function, f, then are $a = \bar{3}$ and $b = \bar{1}$. After matching the Succ branch of the **fold**, we introduce n with a value of $\bar{2}$, since $a = \mathsf{Succ}(\bar{2})$.

Variable n is a special variable, which will be called a "recursive" variable from here on. Any variable which is used in the first position of a call to a recursive function is a recursive variable. All recursive variables are player variables. In this case n is used in the first position in a call to recursive function f. Note that the *add* function is not considered recursive in this context, but the f function contained within it is.

Typing restrictions on folds

We briefly offer some intuition to the reader on the importance of opponent types and player types, and particularly how they interact to ensure that every well-typed function in Pola will halt in polynomial time. The matter will be discussed rigorously in section 3.4.

Figure 3.3 gives two well-typed functions, *add* and *mul*, which naturally define addition and multiplication on the natural numbers that were defined in figure 3.2. The *add* function is exactly as it was defined in figure 3.2, where addition is performed via repeatedly taking the successor. Should we make a call to $add(\bar{6}, \bar{2})$, tracing through the recursive calls in the fold we find that we recurse 6 times, thus performing 6 Succ operations and ending up with a value of $\bar{8}$. The *mul* function is defined nearly identically to the *add* function with three modifications: firstly, that the accumulated value begins at $\bar{0}$, as seen on line 8; secondly, and most importantly, that instead of repeatedly performing a Succ construction, the function repeatedly performs an addition operation; thirdly, the second parameter, y, is an opponent variable, not a player variable, which will be explained in the next paragraph. Consequently, should we make a call to $mul(\bar{6}, \bar{2})$, we would again recurse 6 times, performing 6 addition operations, each time adding $\bar{2}$, yielding a final value of $\overline{12}$.

The reason that the parameter y could be a player variable in the case of *add* is that y is not used to drive a recursion. However, in the case of *mul*, the parameter y is used as the first argument to the function *add*, necessarily an opponent argument. The parameter y in the case of *mul* is (indirectly via *add*) driving a recursion, and thus the typing system enforces that it be an opponent variable. Because both parameters x and y are used to drive recursions in the case of *mul*, both must be opponent variables. We reiterate at

```
1 | exp = x, y |  .fold f(a, b) as {
2 |     Zero.b;
3 |     Succ(− | n).f(n, mul(y, b)) }
4 | in f(x, Succ(Zero));
```

Figure 3.4: A failed attempt at writing an exponential function in Pola. This example will lead to a typing error.

```
1 | z = x |  .fold f(x, y) as {
2 |     Zero.y;
3 |     Succ(− | n).f(n, f(n, Succ(y))) }
4 | in f(x, Zero);
```

Figure 3.5: An exponential function which would yield a typing error in Pola.

this point that every formal parameter to a recursive function—i.e., the parameters a and b in our example—is a player variable. Thus, attempting to rewrite the expression on line 7 to read $f(n, add(b, y))$ (putting b in the first position of the addition) will yield a typing error as b is not an opponent variable but *add*'s first parameter is an opponent parameter.

If arithmetic is performed using the Peano representation of natural numbers given in figure 3.2, it must be impossible to write an exponential function. The reason is that computing an exponential using Peano arithmetic requires an exponential number of Succ constructions. Figure 3.4 gives a natural attempt at writing an exponential function in Pola, following the style of arithmetic functions given in figure 3.3. If addition is iterated successors and multiplication is iterated addition, then exponential is iterated multiplication. The function in figure 3.4 yields a typing error: specifically, the *mul* function requires both arguments to be opponent and, on line 3 of figure 3.4, the variable b is a player variable. The typing system thus enforces that writing an exponential function in such a style is impossible. We will later prove that writing an exponential function is impossible in general.

Duplication of variables—that is, making reference to the same player variable more than once within a single evaluated term—in the player world is disallowed. Consider the program given in figure 3.5. The function z would compute $2^x - 1$ and would operate in time exponential with respect to the size of x, due to the two recursive calls to f. This program would yield a typing error in Pola due to the fact that the variable n is duplicated. n is necessarily a player variable, because all recursive variables are player variables, and there is a general restriction on the duplication of any player variable.

There is one major exception to the restriction on duplication player variables just mentioned, which is that players may be duplicated between branches of a **fold**, an **unfold** (seen in section 3.1.4) and **case** and **peek** constructs (seen in section 3.1.5). For instance, the variable y in figure 3.5 is duplicated within the **fold**: it is used once in the Zero branch and once in the Succ branch of the **fold**, which is permitted. However, the variable n is duplicated within the same function call, which causes the typing violation.

```
1 | data c → Prod(a, b)
2 |     = P1 : c → a
3 |     | P2 : c → b;
```

Figure 3.6: An example of a coinductive data type in Pola, representing the product of two values.

3.1.3 Coinductive data types

Coinductive data types introduce objects which are evaluated or explored lazily. By that, we mean that values within a coinductive object are not computed immediately upon creation of the object, but only when requested—only at the time when the object can be destructed. Consider the example Prod coinductive data type, parameterized over the type variables a and b, given in figure 3.6. Rather than having constructors, coinductive data types have *destructors*[1]. Line 2 gives the first destructor, P_1, of Prod, and gives it type $c \to a$. That is, an object of type $\mathsf{Prod}(a, b)$ can be destructed with the P_1 destructor and yield a value of type a. Line 3 gives the second destructor, P_2, which is similar to the P_1 destructor. The Prod data type captures the data type of pairs: the coinductive data type which holds two values.

For a reader familiar with object-oriented programming, a good intuition for coinductive data types is that of classes. A coinductive data type definition can be thought of as a class, and a destructor can be thought of as a method declaration.

Coinductive values are created by "records", which are terms corresponding to destructors enclosed within parentheses. By convention we put white space on the insides of the parentheses of a record to more clearly distinguish them. An example coinductive value of type $\mathsf{Prod}(\mathsf{Nat}, \mathsf{Bool})$ would be $(\ \mathsf{P}_1 : \bar{3}; \mathsf{P}_2 : \mathsf{False}\ )$. This denotes a coinductive value where the first destructor is bound to term $\bar{3}$ and the second destructor is bound to the term False. Using the object-oriented analogy, this would be analogous to defining an object where invoking the nullary method P_1 would evaluate and return the term $\bar{3}$ and invoking the nullary method P_2 would evaluate and return the term False. Note that the terms within the body of the record—in this example, $\bar{3}$ and False—are not evaluated at the time the record is created, but at the time the record is destructed.

Square bracket notation is used to denote the destruction of a record. The name of the destructor is given first and the record to be destructed is given in square brackets adjacent to that. For instance, the term $\mathsf{P}_1[(\ \mathsf{P}_1 : \bar{3}; \mathsf{P}_2 : \mathsf{False}\ )]$ denotes invoking the P_1 destructor on the given record and would evaluate to the value $\bar{3}$. This would be analogous to invoking a method in object-oriented programming. Note that the destructors ("methods") of Prod are not parametric, though in general destructors may be. If a destructor has parameters, arguments for destruction are given after the square brackets, as can be seen in example 3.7.

Lines 1–2 of figure 3.7 define the Fn coinductive data type, the type of closures. Fn has a single destructor which takes in one parameter, of type a, and returns a value of type b. For example, $\mathsf{Fn}(\mathsf{Bool}, \mathsf{Nat})$ is the type of closures from Booleans to natural numbers.

[1]The usage of the word "destructor" bears no resemblance to the notion of an object destructor in some object-oriented languages such as C++.

```
1 | data c → Fn(a, b)
2 |     = Eval : c, a → b;
3 | apply =   | f, x.Eval[f](x);
```

Figure 3.7: An example of a coinductive data type in Pola.

```
1  | data c → Fn(a, b)
2  |     = Eval : c, a → b;
3  | data c → InfList(a)
4  |     = Head : c → a
5  |     | Tail : c → c;
6  | allNats =   | .unfold g(x) as (
7  |     Head : x;
8  |     Tail : g(Succ(x)) )
9  | in g(Zero);
10 | map =   | f, l.unfold g(l) as (
11 |     Head : Eval[f](Head[l]);
12 |     Tail : g(Tail[l]) )
13 | in g(l);
```

Figure 3.8: Infinite lists in Pola.

Line 3 gives a function, *apply*, which takes in two player parameters: f, a closure of type $\mathsf{Fn}(a, b)$; and x, a value of type a. It then evaluates to the result of destructor f with the value of x, in effect applying the function f to the value x.

An example of using the function *apply* would be in the expression $apply((\ \mathsf{Eval} : x.\mathsf{Succ}(x)\ ), \bar{2})$. The record $(\ \mathsf{Eval} : x.\mathsf{Succ}(x)\ )$ is of type $\mathsf{Fn}(\mathsf{Nat}, \mathsf{Nat})$, a closure taking values from Nat to Nat. The expression within the record, $\mathsf{Succ}(x)$, is evaluated lazily: it is not evaluated until the closure is "destructed". $apply((\ \mathsf{Eval} : x.\mathsf{Succ}(x)\ ), \bar{2})$ binds the value $(\ \mathsf{Eval} : x.\mathsf{Succ}(x)\ )$ to f and $\bar{2}$ to x, giving the term $\mathsf{Eval}[(\ \mathsf{Eval} : x.\mathsf{Succ}(x)\ )](\bar{2})$. We destruct the given record with the Eval destructor, using the value $\bar{2}$ for the bound variable x, and are left with the term $\mathsf{Succ}(\bar{2})$, which yields the returned value of $\bar{3}$.

3.1.4 Unfolds

Because the body of a record is lazy, coinductive data allow the use of infinite data structures. Figure 3.8 gives an infinite data structure, the InfList. Lines 3–5 define the infinite list data structure. The Head destructor given on line 4 evaluates and returns the first element of the list; the Tail destructor given on line 5 returns the tail of the list, which itself is an infinite list.

Because the InfList data type is recursive, namely via the Tail destructor, it is not possible to define an InfList object merely with the record syntax. A recursive construct is required, which is the **unfold** construct. The function *allNats*, given on lines 6–9, defines an infinite list using this construct. Like the **fold** construct, it introduces a

```
1  | data BinaryTree(a) → c
2  |     = Leaf : a → c
3  |     | Node : c, c → c;
4  | data Bool → c
5  |     = True :   → c
6  |     | False :  → c;
7  | isLeafOpp = x |  .case x of {
8  |     Leaf(−).True;
9  |     Node(−, −).False };
10 | isLeafPlayer =  | x.peek x of {
11 |     Leaf(−).True;
12 |     Node(−, −).False };
```

Figure 3.9: Examples of using the **case** and **peek** constructs.

recursive function, in this case g, which can only be used within the **unfold**. Unlike the recursive functions introduced by **fold** constructs, the recursive functions introduced by **unfold** constructs require zero or more parameters, not one or more. In the case of g defined on line 6, it takes one parameter, x, which is necessarily a player variable. On line 9 it is given an initial value of **Zero**. The **unfold** construct itself performs no computation whatsoever; however, if the object it produces is ever destructed, then the expressions given within the body of the **unfold** will be computed accordingly.

The infinite list produced by the *allNats* function has type InfList(Nat), the infinite list of natural numbers. Evaluating Head[*allNats*()]—i.e., performing the Head destruction of the object produced by *allNats*—yields the value $\bar{0}$, since that is the initial value of x. Evaluating Tail[*allNats*()] yields an infinite list, but one where the value of x has been incremented to $\bar{1}$. Evaluating Head[Tail[*allNats*()]] thus yields a value of $\bar{1}$. We see, then, that *allNats* produces the infinite list of all natural numbers, i.e., $[\bar{0}, \bar{1}, \bar{2}, \ldots]$. Elements of the list are only produced so long as we perform Tail destructions, of which there must be finitely many.

The *map* function given in figure 3.8 is a function which takes in two parameters: a closure f representing a function to apply to elements of a list, and an infinite list l. The function produces its own infinite list of elements and is evaluated lazily.

3.1.5 Cases and peeks

When dealing with inductive data, it can be useful to perform pattern-matching outside the setting of a **fold** construct. The **case** and **peek** constructs allow for this, as shown in figure 3.9. Lines 1–3 introduce a binary tree inductive data type, where the leaves of the tree are populated with data but the nodes are not. Lines 4–6 introduce the type of Booleans. Lines 7–12 then introduce two functions which perform the same task: to return a Boolean value of true or false accordingly if the input parameter, a binary tree, is a leaf. The difference between the two functions is whether the parameter is an opponent variable or a player variable. This highlights the sole distinction between the

Named type	*TypeExpr* :=	*typeName*
Type variable	*TypeExpr* :=	*typeVar*
Type application	*TypeExpr* :=	(*TypeExpr TypeExpr*)
Tuple type	*TypeExpr* :=	*TypeExpr* × *TypeExpr*
Opponent type	*TypeExpr* :=	**o**(*TypeExpr*)
Recursive type	*TypeExpr* :=	**r**(*typeVar*, *TypeExpr*)
Branch type	*TypeExpr* :=	**br**(*TypeExpr*)
Co-branch type	*TypeExpr* :=	$\overline{\textbf{br}}$(*TypeExpr*)
Arrow type	*TypeExpr* :=	*TypeExpr* → *TypeExpr*

Figure 3.10: Type expressions in Compositional Pola.

case and **peek** constructs: a **case** construct deals with values in the opponent world and the **peek** construct deals with values in the player world. The patterns on lines 8, 9, 11 and 12 have variables bound to the Leaf and Node constructors being anonymous: this is not a requirement, but is commonplace in **case** and **peek** constructs whenever the data contained within are not used for anything.

3.2 Types

In this section we introduce types, which are of paramount importance to understanding Pola. All of the restrictions which constrain Pola to polynomial-time running times are enforced by the typing system. The use of types and typing will become clearer in future sections, namely section 3.4 and section 3.6. For now we look at the syntax and structure of types in Pola.

3.2.1 Syntax of type expressions

The structure of types in Compositional Pola is given in figure 3.10. Named types refer to concrete types which have been already introduced, such as Nat or Bool. Some types may be parameterized over type variables: for instance, (List Nat) refers to a list of natural numbers and (List α) refers to a list of elements with unknown type. Tuple types represent pairs, which can be composed for more general tuples of any arity. An opponent type gives a typing restriction that the term can be composed only of symbols in the opponent context, a restriction which will be explained in section 3.4. The recursive type is used to control safe recursion inside **fold** and **unfold** constructs, which will be explained in section 3.4.3. Branch and co-branch types are used for terms which have meaning only within inductive case constructors—e.g., **peek** or **case** constructs—or within coinductive constructions—e.g., **record** or **unfold** constructs. Arrow types are used for higher-order terms.

Typical in the course of algorithms and proofs about the type correctness is the reliance on structure in the types. By abstracting over the various forms of type structure we can improve clarity and brevity in the proofs by reducing the number of cases to consider. This will be of primary use when making proofs about the correctness of

Type definition	*TypeDefn* :=	**data** *TypeName* → *typeVar* = *Constructor**;	Inductive type
	TypeDefn :=	**data** *typeVar* → *TypeName* = *Destructor**;	Coinductive type
Type name	*TypeName* :=	*typeName*(*typeVar**)	Parametric type
Constructor	*Constructor* :=	*consName* : *TypeExpr** → *typeVar*	
Destructor	*Destructor* :=	*destName* : *typeVar*, *TypeExpr** → *TypeExpr*	

Figure 3.11: Abstract syntax for type definitions in Compositional Pola.

typing, as we can avoid having to consider each structure of type separately.

Notation 1. *Abstractly we consider that a type has structure* $\mathcal{T}_x(\alpha_1, \ldots, \alpha_n)$ *for some concrete form of structure* x *and parameterized over some types* $\alpha_1, \ldots, \alpha_n$*. Specifically:*

- *A type variable* x *has structure* $\mathcal{T}_v(x)$*;*
- *A named type* X *has structure* $\mathcal{T}_T(X)$*;*
- *A type application* $(\alpha\ \beta)$ *has structure* $\mathcal{T}_a(\alpha, \beta)$*;*
- *A tuple of type* $\alpha \times \beta$ *has structure* $\mathcal{T}_\times(\alpha, \beta)$*;*
- *An opponent type* $\mathbf{o}(\alpha)$ *has structure* $\mathcal{T}_o(\alpha)$*;*
- *A recursive type* $\mathbf{r}(\alpha)$ *has structure* $\mathcal{T}_r(\alpha)$*;*
- *A branch type* $\mathbf{br}(\alpha)$ *has structure* $\mathcal{T}_{br}(\alpha)$*;*
- *A co-branch type* $\bar{\mathbf{br}}(\alpha)$ *has structure* $\mathcal{T}_{\bar{br}}(\alpha)$*;*
- *An arrow type* $\alpha \to \beta$ *has structure* $\mathcal{T}_\to(\alpha, \beta)$*;*

3.2.2 Syntax of type definitions

At the top level, Pola consists of a list of definitions of data types, both inductive and coinductive algebraic data types, and functions. Because general recursion is disallowed in Compositional Pola, the order in which symbols are declared or defined is important. By convention we will show keywords in **bold**, type names, destructors and constructors in sans-serif, and identifiers in *italics*. Specifically, non-terminals will be italicized beginning with an upper-case letter and terminals will be italicized beginning with a lower-case letter.

The syntax for data type definitions is given in figure 3.11. Examples of inductive data types are shown in figure 3.12.

Coinductive data types can be less intuitive due to their relative rarity in functional languages. Coinductive data types are used to store potential computation for a later time, to be "destructed". Examples of coinductive data types are given in figure 3.13.

Natural numbers	1	**data** Nat $\to c$
	2	$=$ Zero : $\to c$
	3	$\mid$ Succ : $c \to c$;
Linked lists	1	**data** List$(a) \to c$
	2	$=$ Nil : $\to c$
	3	$\mid$ Cons : $a, c \to c$;
Binary trees (data on leaves)	1	**data** Tree$(a) \to c$
	2	$=$ Leaf : $a \to c$
	3	$\mid$ Node : $c, c \to c$;
Binary trees (data on internal nodes)	1	**data** Tree2$(a) \to c$
	2	$=$ Leaf2 : $\to c$
	3	$\mid$ Node2 : $c, a, c \to c$;
Optional type	1	**data** Maybe$(a) \to c$
	2	$=$ Nothing : $\to c$
	3	$\mid$ Just : $a \to c$;
Coproduct	1	**data** Either$(a, b) \to c$
	2	$=$ Left : $a \to c$
	3	$\mid$ Right : $b \to c$;

Figure 3.12: Example inductive data types in Pola.

Closure	1	**data** $c \to$ Fn(a, b)
	2	$=$ Eval : $c, a \to b$;
Infinite list	1	**data** $c \to$ InfList(a)
	2	$=$ Head : $c \to a$
	3	$\mid$ Tail : $c \to c$;

Figure 3.13: Example coinductive data types in Pola.

Opponent typing is used exclusively within arrow types to designate that a particular argument to a function may not contain player terms. Branch and co-branch types are used to enforce typing agreement between, respectively: branch constructs (i.e., peek, case, fcase) and their branches; and records and their co-branches. For instance, the keyword peek *per se* has a type of $(\mathbf{br}(\alpha \to \beta) \to \alpha\beta)$ and the terms which perform pattern-matching within the peek have a type of $(\mathbf{br}(\alpha \to \beta)$. In this way, the use of branch and co-branch typing ensures that a matching expression follows a keyword like peek.

3.2.3 Constructor and destructor typing

Associated with each program is a global context, Ξ, which maintains the types of all global symbols, such as constructors and destructors and previously-defined functions. Each data type definition adds to this global context, Ξ.

Consider the following inductive data type definition for type A:

$$\begin{array}{l|l} 1 & \mathbf{data}\ \mathsf{A}(x_1, \ldots, x_n) \to c \\ 2 & \quad = \mathsf{C}_1 : \tau_{1,1}, \ldots, \tau_{1,p_1} \to c \\ 3 & \quad \mid \cdots \\ 4 & \quad \mid \mathsf{C}_m : \tau_{m,1}, \ldots, \tau_{m,p_m} \to c; \end{array}$$

For each constructor, $\mathsf{C}_i : \tau_{i,1}, \ldots, \tau_{i,p_i} \to c$ we add, to Ξ, the following type signature:

$$\mathsf{C}_i : (\tau_{i,1} \times \ldots \times \tau_{i,p_i}) \to \mathsf{A}(x_1, \ldots, x_n)$$

It is taken implicitly that the constructor is parameterized over type variables $x_1, \ldots, x_n$.

For coinductive types, consider the following coinductive data type definition for type B:

$$\begin{array}{l|l} 1 & \mathbf{data}\ c \to \mathsf{B}(x_1, \ldots, x_n) \\ 2 & \quad = \mathsf{D}_1 : c, \tau_{1,1}, \ldots, \tau_{1,p_1} \to \upsilon_1 \\ 3 & \quad \mid \cdots \\ 4 & \quad \mid \mathsf{D}_m : c, \tau_{m,1} \ldots, \tau_{m,p_m} \to \upsilon_m \end{array}$$

For each destructor, $\mathsf{D}_i : c, \tau_{i,1}, \ldots, \tau_{i,p_i} \to \upsilon_i$ we add, to Ξ, the following type signature:

$$\mathsf{D}_i : (\mathsf{B}(x_1, \ldots, x_n) \times \tau_{i,1} \times \cdots \times \tau_{i,p_i}) \to \upsilon_i$$

As with constructors, destructors are parametric over type variables $x_1, \ldots, x_n$.

In our examples, we will typically not include constructors and destructors in the definition of Ξ in the interest of brevity. It is assumed that all constructors and destructors are implicitly in Ξ for data types that are in scope.

3.3 Terms

Here we introduce the terms of Compositional Pola, which are given in a compositional syntax. Later we will see syntactic sugar atop this compositional syntax to define the syntax of Pola, the sugared version of Compositional Pola, as seen in section 3.1. A

Constructor	$Term :=$	$consName$
Destruction	$Term :=$	$destName$
Case	$Term :=$	**case**
Peek	$Term :=$	**peek**
Fold	$Term :=$	**fold**
Fold case	$Term :=$	**fcase**
Function	$Term :=$	$fName$
Tuple pair	$Term :=$	$Term \times Term$
Variable reference	$Term :=$	$varName$
Opponent lambda	$Term :=$	$\lambda^{o} varName.Term$
Player lambda	$Term :=$	$\lambda^{p} varName.Term$
Inductive recursive lambda	$Term :=$	$\lambda^{f} fName.Term$
Coinductive recursive lambda	$Term :=$	$\lambda^{\bar{f}} fName.Term$
Constructor pattern	$Term :=$	$\lambda^{\mathsf{C}} consName.Term$
Variable pattern	$Term :=$	$\lambda^{op} varName\ varName.Term$
Tuple match	$Term :=$	$\Diamond.Term$
Unit tuple match	$Term :=$	$\nabla.Term$
Opponent application	$Term :=$	$(Term\ (Term)_{\mathbf{o}})$
Player application	$Term :=$	$(Term\ Term)$
Branch composition	$Term :=$	$Term + Term$
Unfold	$Term :=$	**unfold**
Record	$Term :=$	**record**
Destructor pattern	$Term :=$	$\lambda^{\mathsf{D}} destName.Term$
Record composition	$Term :=$	$Term \oplus Term$

Figure 3.14: Syntax for terms in Compositional Pola.

program in Compositional Pola consists of a number of type declarations and a number of symbol declarations, where each symbol is bound to a term. The form of compositional Pola terms is given in figure 3.14.

Constructions, destructions and function calls all look similar in Compositional Pola: each is a symbol—either a constructor symbol, destructor symbol or function symbol—juxtaposed with zero or more arguments. Cases and peeks are just a keyword (either **case** or **peek**) juxtaposed with the subject and a branch term, the distinction being whether the subject is allowed to reference player variables (**peek**) or not (**case**). A fold case (**fcase**) is semantically identical to a **peek** but has a typing restriction that allows its use only within a **fold** construct, preventing the duplication of the subject. A branch term is one or more branches composed together with "+", where a branch is a constructor pattern, a tuple match, or a variable pattern. A fold is the keyword **fold** juxtaposed with function lambda term which introduces the recursive function and which includes an **fcase** term in its body. Tuples are built syntactically by composing terms with the "×" symbol. See figure 3.15 for an example of inductive terms in Compositional Pola.

Multiple syntaxes for applications—opponent applications, player applications and higher-order applications—are most useful for type inference, described in section 3.6, where the inference engine can determine up-front whether to perform inference with the

$$
\begin{gathered}
add = \lambda^o x.\lambda^p y.(\lambda^f f.(f\ x_{\mathbf{o}}\ y)) \\
(\mathbf{fold}\ (\lambda^f f.\lambda^p a.\lambda^p b.\mathbf{fcase}\ a \\
(\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.b \\
+\ \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_1\, x_2.\triangledown.(f\ x_2\ (\mathsf{Succ}\ (b \times ()))))))
\end{gathered}
$$

$$
\begin{gathered}
append = \lambda^o l.\lambda^p m.(\lambda^f f.(f\ l_{\mathbf{o}}\ m)) \\
(\mathbf{fold}\ (\lambda^f f.\lambda^p a.\lambda^p z.\mathbf{fcase}\ a \\
(\lambda^{\mathsf{C}}\mathsf{Nil}.\triangledown.z \\
+\ \lambda^{\mathsf{C}}\mathsf{Cons}.\Diamond.\lambda^{op} x_1\, x_3.\Diamond.\lambda^{op} x_2\, x_4.\triangledown.\mathsf{Cons}\ (x_1 \times ((f\ x_4\ z)) \times ())))))
\end{gathered}
$$

Figure 3.15: An example of Compositional Pola syntax, defining two functions, *add* and *append*, which operate on inductive data.

$$
\begin{gathered}
allNats = (\lambda^f g.(g\ (\mathsf{Zero}\ ()))) \\
(\lambda^{\bar{f}} g.\lambda^p x.\mathbf{record}\ (\lambda^{\mathsf{D}}\mathsf{Head}.x \\
\oplus\ \lambda^{\mathsf{D}}\mathsf{Tail}.(g\ (\mathsf{Succ}\ (x \times ())))))
\end{gathered}
$$

$$
\begin{gathered}
map = \lambda^p f.\lambda^p l.(\lambda^f g.(g\ l)) \\
(\lambda^{\bar{f}} g.\lambda^p l.\mathbf{record}\ (\lambda^{\mathsf{D}}\mathsf{Head}.(f\ \mathsf{Eval}\ (l\ \mathsf{Head})) \\
\oplus\ \lambda^{\mathsf{D}}\mathsf{Tail}.(g\ (l\ \mathsf{Tail}))))
\end{gathered}
$$

Figure 3.16: An example of Compositional Pola syntax, defining two functions, *allNats* and *map*, which operate on coinductive data.

player context included or not.

Records—and, by extension, unfolds—are composed of individual "co-branches", where one co-branch is a mapping between a destructor name and the term, as shown in figure 3.16.

3.3.1 Syntactic sugar

To aid in the clarity of the language, we introduce some syntactic sugar for commonly used constructs, which will be used in all future examples in this chapter, and which have already been seen in section 3.1.

- **where** clauses are introduced, binding to either player or opponent variables. t **where** $x = u$ will stand for $(\lambda^p x.t)\ u$ and t **where** $x := u$ will stand for $(\lambda^o x.t)\ u$.

- Tuples are presented in a more conventional way. (t_1, t_2) will stand for $t_1 \times (t_2 \times ())$; (t_1, t_2, t_3) will stand for $t_1 \times (t_2 \times (t_3 \times ()))$; and so on.

- Constructions are presented in a more conventional way and will always be presented in sans-serif typeface. True will stand for $\mathsf{True}\ ()$, the constructor True applied to the unit tuple; $\mathsf{Succ}(n)$ will stand for $\mathsf{Succ}\ (n \times ())$.

- Destructions are presented in a square bracket syntax to clearly distinguish them from constructions, but will also be written in sans-serif typeface. For instance, $\mathsf{Eval}\ (x \times (y \times ()))$ is written as $\mathsf{Eval}[x](y)$.
- Patterns are reformatted to obsolesce the usage of $\Diamond$ and $\triangledown$. $\mathsf{True}.t$ will stand for $\lambda^{\mathsf{C}}\mathsf{True}.\triangledown.t$; $\mathsf{Succ}(x).t$ will stand for $\lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{p}x.\triangledown.t$; and $\mathsf{Cons}(a, b \mid c).t$ will stand for $\lambda^{\mathsf{C}}\mathsf{Cons}.\Diamond.\lambda^{o}a.\Diamond\lambda^{op}bc.\triangledown.t$.
- Branches are grouped together with curly braces and separated by semicolons.
- Co-branches are grouped together with parentheses and separated by semicolons.
- Function applications use a more familiar parenthesis style as compared to juxtaposition, and the usage of subscripted $\mathbf{o}$ is dropped: whether an argument to a function is opponent or player is implied. For instance, $add\ x_{\mathbf{o}}\ y$ is written as $add(x, y)$.

The sugared version is the version more useful for a programmer, and is the version provided by the reference implementation of Pola. For any Pola term, there is a clear and direct corresponding Compositional Pola term. The Compositional Pola notation is given in this document to provide simplicity in some of the rules of inference.

3.4 Typing and well-typedness

We will consider the process of type inference in section 3.6, but first we need to define well-typedness of a Compositional Pola term. The operational semantics given in section 3.5 depend on a term being well-typed.

3.4.1 Contexts and sequents

We consider three contexts to be present at all times: Ξ, the context of global symbols, namely function symbols, constructors and destructors; Γ, the context of opponent variables in scope; and Δ, the context of player symbols in scope.

Each of the contexts, Ξ, Γ, Δ, is a list of mappings from symbols to types. We will consider these contexts to be unordered and without duplication. For instance, some rules may require a split in the player context Δ, written as Δ_1, Δ_2. This splitting of the player context, as in linear logic, enforces that player variables cannot be duplicated. Without loss of generality, we consider that these rules hold for Δ_2, Δ_1 as well, and in cases where at least one of Δ_1 or Δ_2 is empty.

We consider a term to be well-typed if and only if there is at least one type, as defined in section 3.2, associated with a term and there is a proof that the term is of that type, by the sequent rules given in this section. A term may have multiple related types. E.g., the function *length*, to determine the length of a list, has type $length : \mathsf{List}(\mathsf{Nat}) \mid \ \rightarrow \mathsf{Nat}$ and $length : \mathsf{List}(\mathsf{Bool}) \mid \ \rightarrow \mathsf{Nat}$ and an infinite number of other types. The type inference system given in section 3.6 will give the most general type of a term; in the case of *length*, the most general type would be $length : \mathsf{List}(\alpha) \mid \ \rightarrow \mathsf{Nat}$ for all types α.

A rule is of the following form:

$$\frac{\Xi_1 \| \Gamma_1 \mid \Delta_1 \vdash t_1 : \alpha_1 \quad \cdots \quad \Xi_n \| \Gamma_n \mid \Delta_n \vdash t_n : \alpha_n}{\Xi_0 \| \Gamma_0 \mid \Delta_0 \vdash t_0 : \alpha_0}$$

Each $\Xi_i, \Gamma_i, \Delta_i$ is a context as mentioned previously. Each t_i is a Compositional Pola term. Each α_i is a Compositional Pola type. The rule reads that, given the premises, of which there may be zero in the case of an axiom, the Compositional Pola term in the conclusion has the given type, for the given contexts.

3.4.2 Simple terms

Figure 3.17 gives the typing rules for simple terms, ignoring coinductive constructs. Figure 3.18 gives simple rules involving branches and pattern-matching.

3.4.3 Recursive inductive terms

The typing system is the sole mechanism to ensure safe recursion and thus to ensure that all Compositional Pola programs execute within polynomial time. The primary mechanism to accomplish this is the introduction of a type variable which cannot be matched with any other type and which is used to drive recursion. This mechanism prevents a recursive function from being called with an improper parameter. For instance, should one wish to write a Compositional Pola function which recurses over the natural numbers, driving the recursion with the natural number n, it is imperative that only the natural number $n-1$ be used to call the recursive function the next time, and $n-2$ the time after that, and so on. Structurally we mean that driving a recursive function with the term $\mathsf{Succ}(t)$, for some natural number term t in normal form, we must guarantee that one may only be able to recurse using the term t as a parameter to the recursive function.

We accomplish this by ensuring that any variable which is used to call a recursive function has a particular type, a type unique to the recursive function in question, which cannot be matched with any other type. We require a unique type for each recursive function, as opposed to a more obvious solution of using one recursive type per concrete type, to ensure that recursive variables used in nested **fold** constructs refer only to their own respective recursive functions.

Symbol reference	$\dfrac{(x : \alpha) \in \Gamma \text{ or } (x : \alpha) \in \Delta \text{ or } (x : \alpha) \in \Xi}{\Xi \| \Gamma \mid \Delta \vdash x : \alpha}$
Construction	$\dfrac{(\mathsf{C}_i : \beta \to \alpha) \in \Xi \quad \Xi \| \Gamma \mid \Delta \vdash t : \beta}{\Xi \| \Gamma \mid \Delta \vdash \mathsf{C}_i\ t : \alpha}$
Opponent application	$\dfrac{\Xi \| \Gamma \mid \Delta \vdash t : \mathbf{o}(\alpha) \to \beta \quad \Xi \| \Gamma \mid \vdash u : \alpha}{\Xi \| \Gamma \mid \Delta \vdash t\ (u)_{\mathbf{o}} : \beta}$
Player application	$\dfrac{\Xi \| \Gamma \mid \Delta_1 \vdash t : \alpha \to \beta \quad \Xi \| \Gamma \mid \Delta_2 \vdash u : \alpha}{\Xi \| \Gamma \mid \Delta_1, \Delta_2 \vdash t\ u : \beta}$
Tuple unit	$\dfrac{}{\Xi \| \Gamma \mid \Delta \vdash () : ()}$
Tuple composition	$\dfrac{\Xi \| \Gamma \mid \Delta \vdash t : \alpha \quad \Xi \| \Gamma \mid \Delta \vdash u : \beta}{\Xi \| \Gamma \mid \Delta_1, \Delta_2 \vdash t \times u : (\alpha \times \beta)}$
Opponent lambda	$\dfrac{\Xi \| (x : \alpha), \Gamma \mid \Delta \vdash t : \beta}{\Xi \| \Gamma \mid \Delta \vdash \lambda^o x.t : \mathbf{o}(\alpha) \to \beta}$
Player lambda	$\dfrac{\Xi \| \Gamma \mid (x : \alpha), \Delta \vdash t : \beta}{\Xi \| \Gamma \mid \Delta \vdash \lambda^p x.t : \alpha \to \beta}$
Recursive lambda	$\dfrac{\Xi, (x : \alpha \to \beta) \| \Gamma \mid \Delta \vdash t : \gamma \to \beta}{\Xi \| \Gamma \mid \Delta \vdash \lambda^f x.t : \mathbf{r}(\alpha, \gamma) \to \gamma \to \beta}$
Fold-variable lambda	$\dfrac{\Xi \| (x_1 : \alpha), \Gamma \mid (x_2 : \gamma), \Delta \vdash t : \beta}{\Xi \| \Gamma \mid \Delta \vdash \lambda^{op} x_1 x_2.t : \mathbf{r}(\gamma, \alpha) \to \beta}$
Fold-variable non-recursive lambda	$\dfrac{\Xi \| (x : \alpha), \Gamma \mid \Delta \vdash t : \beta}{\Xi \| \Gamma \mid \Delta \vdash \lambda^o x.t : \mathbf{r}(\gamma, \alpha) \to \beta}$

Figure 3.17: Typing rules for simple terms.

$$\frac{}{\Xi \| \Gamma \mid \Delta \vdash \mathbf{peek} : \mathbf{br}(\alpha \to \beta) \to \alpha \to \beta} \quad \text{Peek}$$

$$\frac{}{\Xi \| \Gamma \mid \Delta \vdash \mathbf{case} : \mathbf{br}(\mathbf{o}(\alpha) \to \beta) \to \mathbf{o}(\alpha) \to \beta} \quad \text{Case}$$

$$\frac{\Xi \| \Gamma \mid \Delta \vdash t : \alpha}{\Xi \| \Gamma \mid \Delta \vdash \triangledown.t : () \to \alpha} \quad \text{Unit tuple match}$$

$$\frac{\Xi \| \Gamma \mid \Delta \vdash t : \alpha}{\Xi \| \Gamma \mid \Delta \vdash \triangledown.t : \mathbf{o}(()) \to \alpha} \quad \text{Opponent unit tuple match}$$

$$\frac{\Xi \| \Gamma \mid \Delta \vdash t : \alpha}{\Xi \| \Gamma \mid \Delta \vdash \triangledown.t : \mathbf{r}(\beta, ()) \to \alpha} \quad \text{Recursive unit tuple match}$$

$$\frac{\Xi \| \Gamma \mid \Delta \vdash t : \alpha \to \beta \to \gamma}{\Xi \| \Gamma \mid \Delta \vdash \Diamond.t : (\alpha \times \beta) \to \gamma} \quad \text{Tuple match}$$

$$\frac{\Xi \| \Gamma \mid \Delta \vdash t : \mathbf{o}(\alpha) \to \mathbf{o}(\beta) \to \gamma}{\Xi \| \Gamma \mid \Delta \vdash \Diamond.t : \mathbf{o}(\alpha \times \beta) \to \gamma} \quad \text{Opponent tuple match}$$

$$\frac{\Xi \| \Gamma \mid \Delta \vdash t : \mathbf{r}(\delta, \alpha) \to \mathbf{r}(\delta, \beta) \to \gamma}{\Xi \| \Gamma \mid \Delta \vdash \Diamond.t : \mathbf{r}(\delta, \alpha \times \beta) \to \gamma} \quad \text{Recursive tuple match}$$

$$\frac{(\mathsf{C}_i : \gamma \to \alpha), \Xi \| \Gamma \mid \Delta \vdash t : \gamma \to \beta}{(\mathsf{C}_i : \gamma \to \alpha), \Xi \| \Gamma \mid \Delta \vdash \lambda^{\mathsf{C}} \mathsf{C}_i.t : \mathbf{br}(\alpha \to \beta)} \quad \text{Branch}$$

$$\frac{\Xi \| \Gamma \mid \Delta \vdash t : \mathbf{br}(\alpha) \quad \Xi \| \Gamma \mid \Delta \vdash u : \mathbf{br}(\alpha)}{\Xi \| \Gamma \mid \Delta \vdash t + u : \mathbf{br}(\alpha)} \quad \text{Branch composition}$$

Figure 3.18: Typing rules for simple branches and pattern-matching.

$$\frac{(\mathsf{C}_i : \gamma \to \alpha), \Xi \| \Gamma \mid \Delta \vdash t : \mathbf{r}(\delta, \gamma) \to \beta}{(\mathsf{C}_i : \gamma \to \alpha), \Xi \| \Gamma \mid \Delta \vdash \lambda^{\mathsf{C}} \mathsf{C}_i.t : \mathbf{br}(\mathbf{r}(\delta, \alpha) \to \beta)} \quad \text{Recursive branch}$$

$$\frac{}{\Xi \| \Gamma \mid \Delta \vdash \mathbf{fcase} : \alpha \to \mathbf{br}(\mathbf{r}(\beta, \alpha) \to \gamma) \to \gamma} \quad \text{Fold case}$$

$$\frac{}{\Xi \| \Gamma \mid \Delta \vdash \mathbf{fold} : (\mathbf{r}(\beta, \alpha) \to \alpha \to \gamma) \to \mathbf{o}(\alpha) \to \gamma} \quad \begin{array}{c}\text{where } \beta \text{ does not} \\ \text{match any other} \\ \textbf{fold} \text{ type}\end{array} \quad \text{Fold}$$

Figure 3.19: Typing rules for inductive recursive terms.

$$not = \lambda^{p} y.\mathbf{peek}\ (\lambda^{\mathsf{C}}\mathsf{False}.\triangledown.\mathsf{True}\ () + \lambda^{\mathsf{C}}\mathsf{True}.\triangledown.\mathsf{False}\ ())\ y$$
$$parity = \lambda^{o} z.(\mathbf{fold}\ (\lambda^{f} f.\lambda^{p} x.\mathbf{fcase}\ x$$
$$(\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{True}\ () + \lambda^{\mathsf{C}}\mathsf{Succ}.\lozenge.\lambda^{op} x_1\ x_2.\triangledown.(not\ (f\ x_2)))))$$
$$(z)_{\mathbf{o}}$$

```
1 | not =   | y.peek y of {
2 |     False.True;
3 |     True.False };
4 | parity = z |  .fold f(x) as {
5 |     Zero.True;
6 |     Succ(x1 | x2).not(f(x2)) }
7 | in f(z);
```

Figure 3.20: The *parity* function, which determines the parity of the natural number x, and its auxiliary function *not*, both given in both Pola and Compositional Pola syntaxes.

$$\dfrac{\dfrac{}{\begin{array}{c}(not : \mathsf{Bool} \to \mathsf{Bool}),\\ (f : \alpha \to \mathsf{Bool})\end{array} \parallel \begin{array}{c}(z : \mathsf{Nat}),\\ (x_1 : \mathsf{Nat})\end{array} \mid \vdash not : \mathsf{Bool} \to \mathsf{Bool}} \quad \dfrac{\dfrac{}{\begin{array}{c}\cdots,\\ (f : \alpha \to \mathsf{Bool})\end{array} \parallel \cdots \mid \vdash f : \alpha \to \mathsf{Bool}} \quad \dfrac{}{\cdots \parallel \cdots \mid (x_2 : \alpha) \vdash x_2 : \alpha}}{\begin{array}{c}(not : \mathsf{Bool} \to \mathsf{Bool}),\\ (f : \alpha \to \mathsf{Bool})\end{array} \parallel \begin{array}{c}(z : \mathsf{Nat}),\\ (x_1 : \mathsf{Nat})\end{array} \mid (x_2 : \alpha) \vdash f\ x_2 : \mathsf{Bool}}}{(not : \mathsf{Bool} \to \mathsf{Bool}), (f : \alpha \to \mathsf{Bool}) \parallel (z : \mathsf{Nat}), (x_1 : \mathsf{Nat}) \mid (x_2 : \alpha) \vdash not\ (f\ x_2) : \mathsf{Bool}}$$
$$\dfrac{(not : \mathsf{Bool} \to \mathsf{Bool}), (f : \alpha \to \mathsf{Bool}) \parallel (z : \mathsf{Nat}), (x_1 : \mathsf{Nat}) \mid (x_2 : \alpha) \vdash \nabla.(not\ (f\ x_2)) : \mathbf{r}(\alpha, ()) \to \mathsf{Bool}}{(not : \mathsf{Bool} \to \mathsf{Bool}), (f : \alpha \to \mathsf{Bool}) \parallel (z : \mathsf{Nat}) \mid \vdash \lambda^{op} x_1\, x_2.\nabla.(not\ (f\ x_2)) : \mathbf{r}(\alpha, \mathsf{Nat}) \to \mathbf{r}(\alpha, ()) \to \mathsf{Bool}}$$
$$\dfrac{(not : \mathsf{Bool} \to \mathsf{Bool}), (f : \alpha \to \mathsf{Bool}) \parallel (z : \mathsf{Nat}) \mid \vdash \Diamond.\lambda^{op} x_1\, x_2.\nabla.(not\ (f\ x_2)) : \mathbf{r}(\alpha, \mathsf{Nat} \times ()) \to \mathsf{Bool}}{(not : \mathsf{Bool} \to \mathsf{Bool}), (f : \alpha \to \mathsf{Bool}) \parallel (z : \mathsf{Nat}) \mid \vdash \lambda^{\mathsf{C}} \mathsf{Succ}.\Diamond.\lambda^{op} x_1\, x_2.\nabla.(not\ (f\ x_2)) : \mathbf{br}(\mathbf{r}(\alpha, \mathsf{Nat}) \to \mathsf{Bool})}$$

Figure 3.21: A continuation of the derivation below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdots \parallel (z : \mathsf{Nat}) \mid \vdash \mathsf{True} : () \to \mathsf{Bool}} \quad \dfrac{}{\cdots \parallel (z : \mathsf{Nat}) \mid \vdash () : ()}}{(not : \mathsf{Bool} \to \mathsf{Bool}), (f : \alpha \to \mathsf{Bool}) \parallel (z : \mathsf{Nat}) \mid \vdash \mathsf{True}\ () : \mathsf{Bool}}}{(not : \mathsf{Bool} \to \mathsf{Bool}), (f : \alpha \to \mathsf{Bool}) \parallel (z : \mathsf{Nat}) \mid \vdash \nabla.\mathsf{True}\ () : \mathbf{r}(\alpha, ()) \to \mathsf{Bool}}}{(not : \mathsf{Bool} \to \mathsf{Bool}), (f : \alpha \to \mathsf{Bool}) \parallel (z : \mathsf{Nat}) \mid \vdash \lambda^{\mathsf{C}} \mathsf{Zero}.\nabla.\mathsf{True}\ () : \mathbf{br}(\mathbf{r}(\alpha, \mathsf{Nat}) \to \mathsf{Bool})} \quad \text{Continued in figure 3.21}}{(not : \mathsf{Bool} \to \mathsf{Bool}), (f : \alpha \to \mathsf{Bool}) \parallel (z : \mathsf{Nat}) \mid \vdash \begin{array}{c}\lambda^{\mathsf{C}} \mathsf{Zero}.\nabla.\mathsf{True}\ ()\\ + \lambda^{\mathsf{C}} \mathsf{Succ}.\Diamond.\lambda^{op} x_1\, x_2.\nabla.(not\ (f\ x_2))\end{array} : \mathbf{br}(\mathbf{r}(\alpha, \mathsf{Nat}) \to \mathsf{Bool})}$$

Figure 3.22: A continuation of the derivation below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdots \| \cdots \mid \vdash \mathbf{fcase} : \mathsf{Nat} \to \mathbf{br}\left(\begin{array}{c}\mathbf{r}(\alpha, \mathsf{Nat}) \\ \to \mathsf{Bool}\end{array}\right) \to \mathsf{Bool}} \quad \dfrac{}{\cdots \| \cdots \mid (x : \mathsf{Nat}) \vdash x : \mathsf{Nat}}}{\begin{array}{c}(not : \mathsf{Bool} \to \mathsf{Bool}), \\ (f : \alpha \to \mathsf{Bool})\end{array} \| (z : \mathsf{Nat}) \mid (x : \mathsf{Nat}) \vdash \mathbf{fcase}\ x : \mathbf{br}\left(\begin{array}{c}\mathbf{r}(\alpha, \mathsf{Nat}) \\ \to \mathsf{Bool}\end{array}\right) \to \mathsf{Bool}} \quad \text{Continued in figure 3.22}}{\begin{array}{c}(not : \mathsf{Bool} \to \mathsf{Bool}), \\ (f : \alpha \to \mathsf{Bool})\end{array} \| (z : \mathsf{Nat}) \mid (x : \mathsf{Nat}) \vdash \left(\begin{array}{c}\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\nabla.\mathsf{True}\ () \\ +\ \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_1\, x_2.\nabla.(not\ (f\ x_2)))\end{array}\right) : \mathsf{Bool}}}{\begin{array}{c}(not : \mathsf{Bool} \to \mathsf{Bool}), \\ (f : \alpha \to \mathsf{Bool})\end{array} \| (z : \mathsf{Nat}) \mid \vdash \left(\begin{array}{c}\lambda^{p} x.\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\nabla.\mathsf{True}\ () \\ +\ \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_1\, x_2.\nabla.(not\ (f\ x_2)))\end{array}\right) : \mathsf{Nat} \to \mathsf{Bool}}}{(not : \mathsf{Bool} \to \mathsf{Bool}) \| (z : \mathsf{Nat}) \mid \vdash \left(\begin{array}{c}\lambda^{f} f.\lambda^{p} x.\mathbf{fcase}\ x \\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\nabla.\mathsf{True}\ () \\ +\ \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_1\, x_2.\nabla.(not\ (f\ x_2)))\end{array}\right) : \mathbf{r}(\alpha, \mathsf{Nat}) \to \mathsf{Nat} \to \mathsf{Bool}}$$

Figure 3.23: A continuation of the below derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\cdots \| (z : \mathsf{Nat}) \mid \vdash \mathbf{fold} : (\mathbf{r}(\alpha, \mathsf{Nat}) \to \mathsf{Nat} \to \mathsf{Bool}) \to \mathbf{o}(\mathsf{Nat}) \to \mathsf{Bool}} \quad \text{Continued in figure 3.23}}{(not : \mathsf{Bool} \to \mathsf{Bool}) \| (z : \mathsf{Nat}) \mid \vdash \left(\begin{array}{c}\mathbf{fold}\ (\lambda^{f} f.\lambda^{p} x.\mathbf{fcase}\ x \\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\nabla.\mathsf{True}\ () \\ +\ \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_1\, x_2.\nabla.(not\ (f\ x_2))))\end{array}\right) : \mathbf{o}(\mathsf{Nat}) \to \mathsf{Bool}} \quad \dfrac{}{\cdots \| (z : \mathsf{Nat}) \mid \vdash z : \mathsf{Nat}}}{(not : \mathsf{Bool} \to \mathsf{Bool}) \| (z : \mathsf{Nat}) \mid \vdash \left(\begin{array}{c}(\mathbf{fold}\ (\lambda^{f} f.\lambda^{p} x.\mathbf{fcase}\ x \\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\nabla.\mathsf{True}\ () + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_1\, x_2.\nabla.(not\ (f\ x_2))))) \\ (z)_{\mathbf{o}}\end{array}\right) : \mathsf{Bool}}}{(not : \mathsf{Bool} \to \mathsf{Bool}) \| \ \mid \vdash \left(\begin{array}{c}\lambda^{o} z.(\mathbf{fold}\ (\lambda^{f} f.\lambda^{p} x.\mathbf{fcase}\ x \\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\nabla.\mathsf{True}\ () + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_1\, x_2.\nabla.(not\ (f\ x_2))))) \\ (z)_{\mathbf{o}}\end{array}\right) : \mathbf{o}(\mathsf{Nat}) \to \mathsf{Bool}}$$

Figure 3.24: A type derivation of the *parity* function given in figure 3.20.

Figure 3.19 gives the typing rules for recursive terms. Consider the *parity* function given in figure 3.20. Figure 3.24 gives a full type derivation of the function, resulting in a correct typing of $parity : \mathbf{o}(\mathsf{Nat}) \to \mathsf{Bool}$, proving that the *parity* function is well-typed.

3.4.4 Coinductive terms

Figure 3.25 gives the rules for coinductive terms in Compositional Pola. There is no rule for destruction as this is just a special case of symbol application. There are effectively two kinds of coinductive terms: non-recursive records and recursive records (unfolds). Recursive coinductive data types may have non-recursive destructors to them, which necessitates non-recursive co-branches of recursive records.

Figure 3.29 gives a complete typing derivation of the *allNats* function given in figure 3.26.

$$\frac{\Xi\|\Gamma \mid \Delta \vdash t : \beta}{\Xi\|\Gamma \mid \Delta \vdash \lambda^{\mathsf{D}}\mathsf{D}_i.t : \bar{\mathbf{br}}(\alpha)} \text{ where } (\mathsf{D}_i : \alpha \to \beta) \in \Xi$$

Non-recursive co-branch

$$\frac{\Xi\|\Gamma \vdash t : \beta}{\Xi\|\Gamma \mid \Delta \vdash \lambda^{\mathsf{D}}\mathsf{D}_i.t : \bar{\mathbf{br}}(\mathbf{r}(\delta, \alpha))} \text{ where } (\mathsf{D}_i : \alpha \to \beta) \in \Xi, \ \alpha \neq \beta$$

Non-recursive co-branch of recursive record

$$\frac{\Xi\|\Gamma \mid \Delta \vdash t : \delta}{\Xi\|\Gamma \mid \Delta \vdash \lambda^{\mathsf{D}}\mathsf{D}_i.t : \bar{\mathbf{br}}(\mathbf{r}(\delta, \alpha))} \text{ where } (\mathsf{D}_i : \alpha \to \alpha) \in \Xi$$

Recursive co-branch

$$\frac{\Xi\|\Gamma \mid \Delta \vdash t : \bar{\mathbf{br}}(\alpha) \quad \Xi\|\Gamma \mid \Delta \vdash u : \bar{\mathbf{br}}(\alpha)}{\Xi\|\Gamma \mid \Delta \vdash t \oplus u : \bar{\mathbf{br}}(\alpha)}$$

Co-branch composition

$$\frac{(g : \alpha), \Xi\|\Gamma \mid \Delta \vdash \lambda^{\bar{f}} g.t : \mathbf{r}(\alpha, \beta) \to \gamma}{\Xi\|\Gamma \mid \Delta \vdash \lambda^{\bar{f}} g.t : \mathbf{r}(\alpha, \beta) \to \gamma} \text{ where } (g : \zeta) \notin \Xi$$

Coinductive recursive lambda introduction

$$\frac{(g : \delta \to \beta), \Xi\|\Gamma \mid \Delta \vdash \lambda^{\bar{f}} g.t : \mathbf{r}(\alpha, \epsilon) \to \gamma}{(g : \beta), \Xi\|\Gamma \mid \Delta \vdash \lambda^{\bar{f}} g.t : \mathbf{r}(\alpha, \delta \to \epsilon) \to\to \gamma}$$

Coinductive recursive lambda with parameter

$$\frac{(g : \beta), \Xi\|\Gamma \mid \Delta \vdash t : \beta[\gamma/\alpha]}{(g : \beta), \Xi\|\Gamma \mid \Delta \vdash \lambda^{\bar{f}} g.t : \mathbf{r}(\alpha, \delta) \to \gamma}$$

Coinductive recursive lambda without parameter

$$\frac{}{\Xi\|\Gamma \mid \Delta \vdash \mathbf{record} : \bar{\mathbf{br}}(\alpha) \to \alpha}$$

Record

$$\frac{}{\Xi\|\Gamma \mid \Delta \vdash \mathbf{unfold} : (\mathbf{r}(\alpha, \beta) \to \bar{\mathbf{br}}(\mathbf{r}(\alpha, \beta))) \to \beta}$$

Nullary unfold

$$\frac{\Xi\|\Gamma \mid \Delta \vdash \mathbf{unfold} : (\mathbf{r}(\alpha, \gamma) \to \delta) \to \delta}{\Xi\|\Gamma \mid \Delta \vdash \mathbf{unfold} : (\mathbf{r}(\alpha, \beta \to \gamma) \to \beta \to \delta) \to \mathbf{o}(\beta) \to \delta}$$

Unfold with parameter

Figure 3.25: Typing rules for coinductive terms.

$$allNats = \mathbf{unfold}\ (\lambda^{\bar{f}} g.\lambda^{p} x.(\lambda^{\mathsf{D}}\mathsf{Head}.x \oplus \lambda^{\mathsf{D}}\mathsf{Tail}.g\ (\mathsf{Succ}\ (x \times ()))))\ (\mathsf{Zero}\ ())_{\mathbf{o}}$$

```
1 | data c → InfList(a)
2 |    =  Head : c → a
3 |    |  Tail : c → c;
4 | allNats =   |  .unfold g(x) as (
5 |    Head.x;
6 |    Tail.g(Succ(x)) )
7 | in g(Zero);
```

Figure 3.26: The *allNats* function, which produces the infinite list of all natural numbers.

$$\dfrac{\dfrac{}{(g : \mathsf{Nat} \to \alpha)\| \ | \ \vdash g : \mathsf{Nat} \to \alpha} \quad \dfrac{\dfrac{}{\cdots \vdash \mathsf{Succ} : (\mathsf{Nat} \times ()) \to \mathsf{Nat}} \quad \dfrac{\dfrac{}{\cdots \mid (x : \mathsf{Nat}) \vdash x : \mathsf{Nat}} \quad \dfrac{}{\cdots \vdash () : ()}}{(g : \mathsf{Nat} \to \alpha)\| \ | \ (x : \mathsf{Nat}) \vdash x \times () : \mathsf{Nat} \times ()}}{(g : \mathsf{Nat} \to \alpha)\| \ | \ (x : \mathsf{Nat}) \vdash \mathsf{Succ}\ (x \times ()) : \mathsf{Nat}}}{(g : \mathsf{Nat} \to \alpha)\| \ | \ (x : \mathsf{Nat}) \vdash g\ (\mathsf{Succ}\ (x \times ())) : \alpha}$$
$$(g : \mathsf{Nat} \to \alpha)\| \ | \ (x : \mathsf{Nat}) \vdash \lambda^{\mathsf{D}}\mathsf{Tail}.g\ (\mathsf{Succ}\ (x \times ())) : \bar{\mathbf{br}}(\mathbf{r}(\alpha, \mathsf{InfList}(\mathsf{Nat})))$$

Figure 3.27: A continuation of the below derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(g : \mathsf{Nat} \to \alpha)\| \ | \ (x : \mathsf{Nat}) \vdash x : \mathsf{Nat}}}{(g : \mathsf{Nat} \to \alpha)\| \ | \ (x : \mathsf{Nat}) \vdash \lambda^{\mathsf{D}}\mathsf{Head}.x : \bar{\mathbf{br}}(\mathbf{r}(\alpha, \mathsf{InfList}(\mathsf{Nat})))} \quad \text{Continued in figure 3.27}}{(g : \mathsf{Nat} \to \alpha)\| \ | \ (x : \mathsf{Nat}) \vdash \lambda^{\mathsf{D}}\mathsf{Head}.x \oplus \lambda^{\mathsf{D}}\mathsf{Tail}.g\ (\mathsf{Succ}\ (x \times ())) : \bar{\mathbf{br}}(\mathbf{r}(\alpha, \mathsf{InfList}(\mathsf{Nat})))}}{(g : \mathsf{Nat} \to \alpha)\| \ | \ \vdash \lambda^{p}x.(\lambda^{\mathsf{D}}\mathsf{Head}.x \oplus \lambda^{\mathsf{D}}\mathsf{Tail}.g\ (\mathsf{Succ}\ (x \times ()))) : \mathsf{Nat} \to \bar{\mathbf{br}}(\mathbf{r}(\alpha, \mathsf{InfList}(\mathsf{Nat})))}}{(g : \mathsf{Nat} \to \alpha)\| \ | \ \vdash \lambda^{\bar{f}}g.\lambda^{p}x.(\lambda^{\mathsf{D}}\mathsf{Head}.x \oplus \lambda^{\mathsf{D}}\mathsf{Tail}.g\ (\mathsf{Succ}\ (x \times ()))) : \mathbf{r}(\alpha, \mathsf{InfList}(\mathsf{Nat})) \to \mathsf{Nat} \to \bar{\mathbf{br}}(\mathbf{r}(\alpha, \mathsf{InfList}(\mathsf{Nat})))}}{(g : \alpha)\| \ | \ \vdash \lambda^{\bar{f}}g.\lambda^{p}x.(\lambda^{\mathsf{D}}\mathsf{Head}.x \oplus \lambda^{\mathsf{D}}\mathsf{Tail}.g\ (\mathsf{Succ}\ (x \times ()))) : \mathbf{r}(\alpha, \mathsf{Nat} \to \mathsf{InfList}(\mathsf{Nat})) \to \mathsf{Nat} \to \bar{\mathbf{br}}(\mathbf{r}(\alpha, \mathsf{InfList}(\mathsf{Nat})))}$$
$$\| \ | \ \vdash \lambda^{\bar{f}}g.\lambda^{p}x.(\lambda^{\mathsf{D}}\mathsf{Head}.x \oplus \lambda^{\mathsf{D}}\mathsf{Tail}.g\ (\mathsf{Succ}\ (x \times ()))) : \mathbf{r}(\alpha, \mathsf{Nat} \to \mathsf{InfList}(\mathsf{Nat})) \to \mathsf{Nat} \to \bar{\mathbf{br}}(\mathbf{r}(\alpha, \mathsf{InfList}(\mathsf{Nat})))$$

Figure 3.28: A continuation of the below derivation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdots \vdash \mathbf{unfold} : \begin{pmatrix} \mathbf{r}(\alpha, \mathsf{InfList(Nat)}) \\ \to \bar{\mathbf{b}}\mathbf{r}(\mathbf{r}(\alpha, \mathsf{InfList(Nat)})) \end{pmatrix} \to \mathsf{InfList(Nat)}}}{\cdots \vdash \mathbf{unfold} : \begin{pmatrix} \mathbf{r}(\alpha, \mathsf{Nat} \to \mathsf{InfList(Nat)}) \\ \to \mathsf{Nat} \\ \to \bar{\mathbf{b}}\mathbf{r}(\mathbf{r}(\alpha, \mathsf{InfList(Nat)})) \end{pmatrix} \to \mathbf{o}(\mathsf{Nat}) \to \mathsf{InfList(Nat)}} \quad \text{Continued in figure 3.28}}{\| \mid \vdash \mathbf{unfold}\ (\lambda^{\bar{f}} g.\lambda^{p} x.(\lambda^{\mathsf{D}}\mathsf{Head}.x \oplus \lambda^{\mathsf{D}}\mathsf{Tail}.g\ (\mathsf{Succ}\ (x \times ()))))) : \mathbf{o}(\mathsf{Nat}) \to \mathsf{InfList(Nat)}} \qquad \dfrac{\dfrac{}{\cdots \vdash \mathsf{Zero} : () \to \mathsf{Nat}} \quad \dfrac{}{\cdots \vdash () : ()}}{\| \mid \vdash \mathsf{Zero}\ () : \mathsf{Nat}}}{\| \mid \vdash \mathbf{unfold}\ (\lambda^{\bar{f}} g.\lambda^{p} x.(\lambda^{\mathsf{D}}\mathsf{Head}.x \oplus \lambda^{\mathsf{D}}\mathsf{Tail}.g\ (\mathsf{Succ}\ (x \times ()))))\ (\mathsf{Zero}\ ())_{\mathbf{o}} : \mathsf{InfList(Nat)}}$$

Figure 3.29: A type derivation of the function *allNats* given in figure 3.26.

Arguments to an **unfold** construct are necessarily in the opponent world. Allowing player-world variables into an **unfold** construct can allow super-polynomial functions. The hypothetical exponential function given in figure 3.30 demonstrates this. We define two functions: *exp*, which computes the infinite list of all multiples of 2^x for the parameter x; and the function *expX* which computes just 2^x. E.g., *exp* 0 computes the infinite list $[1, 2, 3, 4, \ldots]$, the list of all multiples of 2^0. By selecting every other element of that list we arrive at *exp* 1 which yields the infinite list $[2, 4, 6, 8, \ldots]$, the list of all multiples of 2^1; if $[2^x, 2 \cdot 2^x, 3 \cdot 2^x, 4 \cdot 2^x, \ldots]$ is the infinite list of multiples of 2^x, then selecting every other element yields $[2 \cdot 2^x, 4 \cdot 2^x, 6 \cdot 2^x, 8 \cdot 2^6, \ldots]$ which is the same as $[2^{x+1}, 2 \cdot 2^{x+1}, 3 \cdot 2^{x+1}, 4 \cdot 2^{x+1}, \ldots]$. The *expX* function then simply returns the first element of one of these generated infinite lists, thus returning the value 2^x.

The complication in this case is the coda to the **unfold** in the Succ case, namely $(\lambda^f g.(g\ (f\ x_2)))$. In effect, $f(n) = g(f(n-1))$ where f is the inductive **fold** recursive function and g is the coinductive unfold recursive function. As a consequence, $f(n) = g^{(n)}(z)$ where z is the record defined in the Zero case; i.e., $z = (\lambda^f g.(g\ (\mathsf{Zero}\ ())))$ $(\lambda^f g.\lambda^p y.(\lambda^{\mathsf{D}}\mathsf{Head}.y \oplus \lambda^{\mathsf{D}}\mathsf{Tail}.(g\ (\mathsf{Succ}\ (y \times ()))))$. For example, $f(3) = g(g(g(z)))$. In the definition of the Tail branch of the record defined in the Succ branch of the **fold**, we apply the Tail destructor twice, effectively skipping over every other element in the parameterized list. Since the g function is being applied to itself n times, we end up skipping over 2^n elements in the Tail destructor.

In this case the typing error is on line 9, where the coda of the **unfold** reads $g(f(x_2))$. x_2 is a player variable and thus this term cannot be typed. In order for an **unfold** to be typed correctly, the coda must consist only of opponent variables.

3.4.5 Summary

At this point, we have provided a programming language which offers typing rules that constrain programs to halt within polynomial time, though the latter has not been proved yet. Before we can make the principal claims of this thesis, we need operational semantics for the language, described in section 3.5 and describe a novel type-inference system, described in section 3.6.

3.5 Operational semantics

The small-step operational semantics are given in figure 3.31 and in figure 3.32. In practice this style of semantics would involve adding values for variables to a store, as opposed to performing substitutions, though for clarity we present in substitution style. The notation $t[u/x]$ is used to stand for the term t with the free occurrences of x replaced by term u.

Due to the fact that we are constraining ourselves to well-typed programs, there are some syntactic cases that need not be considered in the operational semantics. These are as follows:

- For the "constructor match" and "constructor non-match" rules, we are guaranteed that the term we are matching against will eventually have a match in the branches,

$$exp = \lambda^o x.\mathbf{fold}\ (\lambda^{\bar{f}} f.\lambda^p x.\mathbf{fcase}\ x$$
$$(\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathbf{unfold}\ (\lambda^f g.\lambda^p y.(\lambda^{\mathsf{D}}\mathsf{Head}.y \oplus \lambda^{\mathsf{D}}\mathsf{Tail}.g\ (\mathsf{Succ}\ (y \times ()))))\ (\mathsf{Succ}\ (\mathsf{Zero}\ ())))$$
$$+\lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_1\ x_2.\triangledown.\mathbf{unfold}$$
$$(\lambda^{\bar{f}} g.\lambda^p y.(\lambda^{\mathsf{D}}\mathsf{Head}.(y\ \mathsf{Tail})\ \mathsf{Head}$$
$$\oplus\lambda^{\mathsf{D}}\mathsf{Tail}.g\ ((y\ \mathsf{Tail})\ \mathsf{Tail}))\ (f\ x_2))))\ (x)_{\mathbf{o}}$$
$$expX = \lambda^o x.((exp\ x)\ \mathsf{Head})$$

```
1  | exp = x |  .fold f(x) as {
2  |     Zero.unfold g(y) as (
3  |           Head : y;
4  |           Tail : g(Succ(y)) )
5  |         in g(Succ(Zero));
6  |     Succ(x1 | x2).unfold g(y) as (
7  |           Head : Head[Tail[y]];
8  |           Tail : g(Tail[Tail[y]]) )
9  |         in g(f(x2)) }
10 | in f(x);
11 | expX = x |  .Head[exp(x)];
```

Figure 3.30: A hypothetical exponential function which does not rely on duplication of player variables. *expX* would compute the value 2^x for any natural number x if it were able to be typed.

$$\frac{t \Rightarrow t'}{\mathsf{C}\ t \Rightarrow \mathsf{C}\ t'} \quad \text{Constructor}$$

$$\frac{t \Rightarrow t' \quad b\,t' \Rightarrow b'}{\mathbf{case}\ (t)_{\mathbf{o}}\ b \Rightarrow b'} \quad \text{Case}$$

$$\frac{t \Rightarrow t' \quad b\,t' \Rightarrow b'}{\mathbf{peek}\ t\ b \Rightarrow b'} \quad \text{Peek}$$

$$\frac{t \Rightarrow t' \quad b\,t' \Rightarrow b'}{\mathbf{fcase}\ t\ b \Rightarrow b'} \quad \text{FCase}$$

$$\frac{t \Rightarrow \lambda^f f.u \quad u[\mathbf{fold}\ (\lambda^f f.u)/f] \Rightarrow u'}{\mathbf{fold}\ t \Rightarrow u'} \quad \text{Fold}$$

$$\frac{t \Rightarrow \lambda^o x.t' \quad u \Rightarrow u' \quad t'[u'/x] \Rightarrow v}{t\ (u)_{\mathbf{o}} \Rightarrow v} \quad \text{Opponent application}$$

$$\frac{t \Rightarrow \lambda^p x.t' \quad u \Rightarrow u' \quad t'[u'/x] \Rightarrow v}{t\ u \Rightarrow v} \quad \text{Player application}$$

$$\frac{t \Rightarrow (\mathsf{C}_i\ x_i) \quad u_i\,x_i \Rightarrow u_i'}{(\lambda^{\mathsf{C}}\mathsf{C}_i.u_i + b)\,t \Rightarrow u_i'} \quad \text{Constructor match}$$

$$\frac{t \Rightarrow (\mathsf{C}_i\ x_i) \quad b\,(\mathsf{C}_i\,x_i) \Rightarrow u_i'}{(\lambda^{\mathsf{C}}\mathsf{C}_j.u_j + b)\,t \Rightarrow u_i} \text{ where } \mathsf{C}_i \neq \mathsf{C}_j \quad \text{Constructor non-match}$$

$$\frac{t \Rightarrow \lambda^{op} x_1 x_2.t' \quad u \Rightarrow u' \quad t'[u'/x_1, u'/x_2] \Rightarrow v}{t\ u \Rightarrow v} \quad \text{Variable match}$$

$$\frac{t \Rightarrow () \quad u \Rightarrow u'}{(\nabla.u)\ t \Rightarrow u'} \quad \text{Unit tuple match}$$

$$\frac{t \Rightarrow (t_1 \times t_2) \quad (u\ t_1)\ t_2 \Rightarrow u'}{(\Diamond.u)\ t \Rightarrow u'} \quad \text{Tuple match}$$

Figure 3.31: Operational semantics for inductive terms in compositional Pola.

$$\frac{t \Rightarrow \lambda^f g.u \quad u[\lambda^f g.u/g] \Rightarrow u'}{\mathbf{unfold}\, t \Rightarrow u'} \qquad \text{Unfold}$$

$$\frac{u_i \Rightarrow u_i'}{\mathsf{D}_i\,(\lambda^{\mathsf{D}}\mathsf{D}_i.u_i \oplus b) \Rightarrow u_i'} \qquad \text{Destruction match}$$

$$\frac{\mathsf{D}_i\, b \Rightarrow u_i'}{\mathsf{D}_i\,(\lambda^{\mathsf{D}}\mathsf{D}_j.u_j \oplus b) \Rightarrow u_i'} \qquad \text{Destruction non-match}$$

Figure 3.32: Operational semantics for coinductive terms in compositional Pola.

due to typing restrictions.

- Similarly, for the "destructor match" and "destructor non-match" rules, we are guaranteed to find a match in the record we are destructing.

- For the "unit tuple match" and "tuple match" rules, we are guaranteed that the tuple we are matching against will have the appropriate number of elements to be matched.

The following shows the computation $1 + 2$ in an equivalent small-step operational semantics. Annotations for opponent or player arguments—e.g., $x_{\mathbf{o}}$ or x—are elided for clarity. The operational semantics first reduce the problem of $1 + 2$ to the addition of $0 + 3$, as follows:

$$\begin{array}{ll}
 & (\mathbf{fold}\ (\lambda^f f.\lambda^p a.\lambda^p b.\mathbf{fcase}\ a\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.b \\
 & \quad +\lambda^{\mathsf{C}}\mathsf{Succ}.\diamondsuit.\lambda^{op}x_1\,x_2.\triangledown.f\ x_2\ (\mathsf{Succ}\ (b \times ()))))))\ \bar{1}\ \bar{2} \\
\Rightarrow & (\lambda^p a.\lambda^p b.\mathbf{fcase}\ a\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.b \\
 & \quad +\lambda^{\mathsf{C}}\mathsf{Succ}.\diamondsuit.\lambda^{op}x_1\,x_2.\triangledown.(\mathbf{fold}\cdots)\ x_2\ (\mathsf{Succ}\ (b \times ())))))\ \bar{1}\ \bar{2} \\
\Rightarrow & (\lambda^p b.\mathbf{fcase}\ \bar{1}\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.b + \lambda^{\mathsf{C}}\mathsf{Succ}.\diamondsuit.\lambda^{op}x_1\,x_2.\triangledown.(\mathbf{fold}\cdots)\ x_2\ (\mathsf{Succ}\ (b \times ())))))\ \bar{2} \\
\Rightarrow & \mathbf{fcase}\ (\mathsf{Succ}\ ((\mathsf{Zero}\ ()) \times ()))\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\bar{2} \\
 & \quad +\lambda^{\mathsf{C}}\mathsf{Succ}.\diamondsuit.\lambda^{op}x_1\,x_2.\triangledown.(\mathbf{fold}\cdots)\ x_2\ (\mathsf{Succ}\ (\bar{2} \times ()))) \\
\Rightarrow & (\mathsf{Succ}\ ((\mathsf{Zero}\ ()) \times ()))\ (\lambda^{\mathsf{C}}\mathsf{Succ}.\diamondsuit.\lambda^{op}x_1\,x_2.\triangledown.(\mathbf{fold}\cdots)\ x_2\ (\mathsf{Succ}\ (\bar{2} \times ()))) \\
\Rightarrow & (\diamondsuit.\lambda^{op}x_1\,x_2.\triangledown.(\mathbf{fold}\cdots)\ x_2\ (\mathsf{Succ}\ (\bar{2} \times ())))\ ((\mathsf{Zero}\ ()) \times ()) \\
\Rightarrow & ((\lambda^{op}x_1\,x_2.\triangledown.(\mathbf{fold}\cdots)\ x_2\ (\mathsf{Succ}\ (\bar{2} \times ()))))\ (\mathsf{Zero}\ ()))\ () \\
\Rightarrow & (\triangledown.(\mathbf{fold}\cdots)\ \bar{0}\ (\mathsf{Succ}\ (\bar{2} \times ())))\ () \\
\Rightarrow & (\mathbf{fold}\ (\lambda^f f.\lambda^p a.\lambda^p b.\mathbf{fcase}\ a\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.b \\
 & \quad +\lambda^{\mathsf{C}}\mathsf{Succ}.\diamondsuit.\lambda^{op}x_1\,x_2.\triangledown.f\ x_2\ (\mathsf{Succ}\ (b \times ()))))\ \bar{0}\ \bar{3}
\end{array}$$

At this stage we have reduced the problem of computing $1 + 2$ to the problem of computing $0 + 3$. We have performed one recursion to make this reduction. The rest of the computation, reducing the problem of $0 + 3$ to the value of 3, requires no further recursion. The following shows this computation in the big-step operational semantics.

$$
\begin{array}{ll}
\Rightarrow & (\lambda^p a.\lambda^p b.\mathbf{fcase}\ a\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.b \\
& \quad +\lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op}x_1\,x_2.\triangledown.(\mathbf{fold}\cdots)\ x_2\ (\mathsf{Succ}\ (b\times()))))\ \bar{0}\ \bar{3} \\
\Rightarrow & (\lambda^p b.\mathbf{fcase}\ \bar{0}\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.b + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op}x_1\,x_2.\triangledown.(\mathbf{fold}\cdots)\ x_2\ (\mathsf{Succ}\ (b\times()))))\ \bar{3} \\
\Rightarrow & \mathbf{fcase}\ (\mathsf{Zero}\ ())\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\bar{3} \\
& \quad +\lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op}x_1\,x_2.\triangledown.(\mathbf{fold}\cdots)\ x_2\ (\mathsf{Succ}\ (\bar{3}\times())) \\
\Rightarrow & (\triangledown.\bar{3})\ () \\
\Rightarrow & \bar{3}
\end{array}
$$

3.6 Type inference

We present here a type inference system, novel with respect to the fact that it works with the constructs of Compositional Pola, most important of which being the universal types required to safely type **fold** constructs.

Type inference is an important component to any programming language, removing from the programmer the burden of annotating the code with typing information. The goal of a type inference system is that, with a minimum of annotations, the system should efficiently infer the most general type for any term. The rules of type inference given in this section will, in many cases, match quite closely the rules given in section 3.4; however, there is a restriction on the format of a rule of inference which is required to reflect the fact that we may not know any concrete types until type inference and unification have completed.

As is usual in type inference, the process is broken into two stages: the collection of type equations followed by the unification of those type equations.

3.6.1 Sequents

There are three forms of sequents used in type inference:

- $\Xi \| \Gamma \mid \Delta \vdash t : \alpha$, which infers the type of term t where Ξ, Γ, Δ are contexts and α is a type variable;
- $\Xi \| \Gamma \mid \Delta \vdash \overline{t}^o : \alpha$, which infers the pattern-matching term t in the context of a **case** construct; and
- $\Xi \| \Gamma \mid \Delta \vdash \overline{t}^r : \alpha$, which infers the pattern-matching term t in the context of an **fcase** construct.

It is required that α is always a type variable, never a concrete type. Typing constraints are introduced by generating type equations, which appear on the right-hand side of a rule of inference. They are generally of the form $(\alpha \rhd \Psi)$ where α is a possibly-empty list of fresh type variables, guaranteed not to match any other type variable introduced and where Ψ is a possibly-empty list of type equations. The complete structure of type equations is given in section 3.6.4. All type equations have a type variable—never a concrete type—on the left-hand side of the equal sign, which can be quantified.

The usage of $\overline{t}^o$ and $\overline{t}^r$ only has consequence for terms of the form $\Diamond.t$, $\nabla.t$, $t + u$ and $\lambda^{\mathsf{C}}\mathsf{C}_i.t$, where the typing equations generated for those four term structures become slightly different. Specifically, the usage of $\overline{t}^o$ and $\overline{t}^r$ indicates whether the pattern ultimately is to be of an opponent type ($\overline{t}^o$), a recursive type ($\overline{t}^r$) or a player type (unadorned t).

The type inference system used in Pola has two additional unique properties that distinguish it from an ordinary Hindley-Milner type inference system: the restrictions on duplication in the player world; and the use of a "universal type" which cannot be matched with any other type.

3.6.2 Inductive terms

Figure 3.33 gives rules of inference for basic Compositional Pola terms. The variable reference rule of inference says that if a term is associated with type variable α, that α must be constrained to be equal to the type of the variable in the appropriate context. The application and lambda rules set up arrow-type constraints.

Figure 3.34 gives rules of inference for pattern-matching in inductive—i.e., **fold**, **fcase**, **case** and **peek**—constructs. The case rule requires no split in its contexts because the subject of the case, t, must consist only of opponent variables. In the peek rule, however, we must split the player context into Δ_1, the context used for the subject t and Δ_2, the context used for the body of the **peek**. The branch composition rules require that all branches be of the same type. The constructor match rules ensure that variables bound within the patterns match the type of data type being matched. For instance, if the subject of a **case** is of type $\mathsf{List}(\mathsf{Nat})$ and there is a pattern $\lambda^{\mathsf{C}}\mathsf{Cons}.\diamond.\lambda^o x.\diamond.\lambda^o xs.\nabla t$, we must ensure that x is of type Nat. In the case of the opponent constructor match rule, δ would be constrained to equal List and γ would be constrained to equal Nat. The unit match rules are used when there are no variables left to be bound in a pattern and the term within the body can be inferred.

Figure 3.37 gives rules of inference for **fold**, **fcase**, **case** and **peek** constructs. Figure 3.36 gives rules of inference for both constructing tuples and performing matching on tuples.

Figure 3.43 gives an example inference of the *parity* function given in figure 3.20. From this inference process we generate a type equation, working from the bottom-up and working from left-to-right in a depth-first manner. The type equation generated by

Variable reference	$\dfrac{(x : \beta) \in \Gamma \text{ or } (x : \beta) \in \Delta \text{ or } (x : \beta) \in \Xi}{\Xi \| \Gamma \mid \Delta \vdash x : \alpha} \rhd \alpha = \beta$
Construction	$\dfrac{(\mathsf{C}_i : \gamma \to \delta) \in \Xi \quad \Xi \| \Gamma \mid \Delta \vdash t : \beta}{\Xi \| \Gamma \mid \Delta \vdash \mathsf{C}_i\ t : \alpha} \beta \rhd \alpha = \delta, \beta = \gamma$
Opponent application	$\dfrac{\Xi \| \Gamma \mid \Delta \vdash t : \beta \quad \Xi \| \Gamma \mid \ \vdash u : \gamma}{\Xi \| \Gamma \mid \Delta \vdash t\ (u)_{\mathbf{o}} : \alpha} \beta, \gamma \rhd \beta = \mathbf{o}(\gamma) \to \alpha$
Player application	$\dfrac{\Xi \| \Gamma \mid \Delta_1 \vdash t : \beta \quad \Xi \| \Gamma \mid \Delta_2 \vdash u : \gamma}{\Xi \Gamma \mid \Delta_1, \Delta_2 \vdash t\ u : \alpha} \beta, \gamma \rhd \beta = \gamma \to \alpha$
Tuple unit	$\dfrac{}{\Xi \| \Gamma \mid \Delta \vdash () : \alpha} \rhd \alpha = ()$
Tuple composition	$\dfrac{\Xi \| \Gamma \mid \Delta_1 \vdash t : \beta \quad \Xi \| \Gamma \mid \Delta_2 \vdash u : \gamma}{\Xi \| \Gamma \mid \Delta_1, \Delta_2 \vdash (t \times u) : \alpha} \beta, \gamma \rhd \alpha = \beta \times \gamma$
Opponent lambda	$\dfrac{\Xi \| (x : \beta), \Gamma \mid \Delta \vdash t : \gamma}{\Xi \| \Gamma \mid \Delta \vdash \lambda^o x.t : \alpha} \beta, \gamma \rhd \alpha = \mathbf{o}(\beta) \to \gamma$
Player lambda	$\dfrac{\Xi \| \Gamma \mid (x : \beta), \Delta \vdash t : \gamma}{\Xi \| \Gamma \mid \Delta \vdash \lambda^p x.t : \alpha} \beta, \gamma \rhd \alpha = \beta \to \gamma$
Recursive lambda	$\dfrac{\Xi, (f : \delta \to \beta) \| \Gamma \mid \Delta \vdash t : \beta}{\Xi \| \Gamma \mid \Delta \lambda^f f.t : \alpha} \beta, \gamma, \delta \rhd \alpha = \mathbf{r}(\delta, \gamma) \to \beta \to \gamma$
Fold variable lambda	$\dfrac{\Xi \| (x_1 : \beta), \Gamma \mid (x_2 : \gamma), \Delta \vdash \overline{t}^r : \delta}{\Xi \| \Gamma \mid \Delta \vdash \overline{\lambda^{op} x_1 x_2 . t}^r : \alpha} \beta, \gamma, \delta \rhd \alpha = \mathbf{r}(\gamma, \beta) \to \delta$

Figure 3.33: Basic rules of type inference.

Peek
$$\frac{\Xi\|\Gamma \mid \Delta_1 \vdash t : \beta \quad \Xi\|\Gamma \mid \Delta_2 \vdash b : \gamma}{\Xi\|\Gamma \mid \Delta_1, \Delta_2 \vdash \mathbf{peek}\ t\ b : \alpha}\ \beta, \gamma \rhd \gamma = \mathbf{br}(\beta \to \alpha)$$

Case
$$\frac{\Xi\|\Gamma \mid \vdash t : \beta \quad \Xi\|\Gamma \mid \Delta \vdash \overline{b}^{o} : \gamma}{\Xi\|\Gamma \mid \Delta \vdash \mathbf{case}\ t\ b : \alpha}\ \beta, \gamma \rhd \gamma = \mathbf{br}(\mathbf{o}(\beta) \to \alpha)$$

Unstructured branch composition
$$\frac{\Xi\|\Gamma \mid \Delta \vdash b_1 : \alpha \quad \Xi\|\Gamma \mid \Delta \vdash b_2 : \beta}{\Xi\|\Gamma \mid \Delta \vdash b_1 + b_2 : \alpha}\ \beta, \gamma, \delta \rhd \begin{array}{c} \alpha = \beta, \\ \alpha = \mathbf{br}(\gamma \to \delta) \end{array}$$

Structured branch composition
$$\frac{\Xi\|\Gamma \mid \Delta \vdash \overline{b_1}^{X} : \alpha \quad \Xi\|\Gamma \mid \Delta \vdash \overline{b_2}^{X} : \alpha}{\Xi\|\Gamma \mid \Delta \vdash \overline{b_1 + b_2}^{X} : \alpha}\ \gamma, \delta \rhd \alpha = \mathbf{br}(\gamma \to \delta)$$

Opponent constructor match
$$\frac{(\mathsf{C}_i : \delta), \Xi\|\Gamma \mid \Delta \vdash \overline{t}^{o} : \beta}{(\mathsf{C}_i : \delta), \Xi\|\Gamma \mid \Delta \vdash \overline{\lambda^{\mathsf{C}}\mathsf{C}_i.t}^{o} : \alpha}\ \beta, \gamma, \eta \rhd \begin{array}{c} \beta = \mathbf{br}(\mathbf{o}(\gamma) \to \eta), \\ \alpha = \mathbf{br}(\mathbf{o}(\delta(\gamma)) \to \eta) \end{array}$$

Recursion constructor match
$$\frac{(\mathsf{C}_i : \delta), \Xi\|\Gamma \mid \Delta \vdash \overline{t}^{r} : \beta}{(\mathsf{C}_i : \delta), \Xi\|\Gamma \mid \Delta \vdash \overline{\lambda^{\mathsf{C}}\mathsf{C}_i.t}^{r} : \alpha}\ \beta, \gamma, \eta, \zeta \rhd \begin{array}{c} \beta = \mathbf{br}(\mathbf{r}(\zeta, \gamma) \to \eta), \\ \alpha = \mathbf{br}(\mathbf{r}(\zeta, \delta(\gamma)) \to \eta) \end{array}$$

Player constructor match
$$\frac{(\mathsf{C}_i : \delta), \Xi\|\Gamma \mid \Delta \vdash t : \beta}{(\mathsf{C}_i : \delta), \Xi\|\Gamma \mid \Delta \vdash \lambda^{\mathsf{C}}\mathsf{C}_i.t : \alpha}\ \beta, \gamma, \eta \rhd \begin{array}{c} \beta = \mathbf{br}(\gamma \to \eta), \\ \alpha = \mathbf{br}(\delta(\gamma) \to \eta) \end{array}$$

Opponent variable match
$$\frac{\Xi\|(x : \beta), \Gamma \mid \Delta \vdash \overline{t}^{o} : \gamma}{\Xi\|\Gamma \mid \Delta \vdash \overline{\lambda^{o}x.t}^{o} : \alpha}\ \beta, \gamma \rhd \alpha = \mathbf{br}(\mathbf{o}(\beta) \to \gamma)$$

Opponent variable match (fold)
$$\frac{\Xi\|(x : \beta), \Gamma \mid \Delta \vdash \overline{t}^{r} : \gamma}{\Xi\|\Gamma \mid \Delta \vdash \overline{\lambda^{o}x.t}^{r} : \alpha}\ \beta, \gamma, \delta \rhd \alpha = \mathbf{br}(\mathbf{r}(\delta, \beta) \to \gamma)$$

Player variable match
$$\frac{\Xi\|(x : \beta), \Delta \vdash t : \gamma}{\Xi\|\Gamma \mid \Delta \vdash \lambda^{p}x.t : \alpha}\ \beta, \gamma \rhd \alpha = \mathbf{br}(\beta \to \gamma)$$

Figure 3.34: Rules of type inference for pattern matching.

Halt player pattern matching	$\dfrac{\Xi \Vert \Gamma \mid \Delta \vdash t : \beta}{\Xi \Vert \Gamma \mid \Delta \vdash \overline{t}^{p} : \alpha}$	where t is not pattern matching $\beta \rhd \alpha = \mathbf{br}(\beta)$
Halt opponent pattern matching	$\dfrac{\Xi \Vert \Gamma \mid \Delta \vdash t : \alpha}{\Xi \Vert \Gamma \mid \Delta \vdash \overline{t}^{o} : \alpha}$	where t is not pattern matching $\beta \rhd \alpha = \mathbf{br}(\beta)$
Halt recursive pattern matching	$\dfrac{\Xi \Vert \Gamma \mid \Delta \vdash t : \beta}{\Xi \Vert \Gamma \mid \Delta \vdash \overline{t}^{r} : \alpha}$	where t is not pattern matching $\beta, \gamma \rhd \alpha = \mathbf{br}(\mathbf{r}(\gamma, \beta))$

Figure 3.35: Rules for halting pattern matching.

Opponent tuple match	$\dfrac{\Xi \Vert \Gamma \mid \Delta \vdash \overline{t}^{o} : \beta}{\Xi \Vert \Gamma \mid \Delta \vdash \overline{\Diamond . t}^{o} : \alpha}\ \beta, \gamma, \delta, \eta \rhd$	$\beta = \mathbf{br}(\mathbf{o}(\gamma) \to \mathbf{br}(\mathbf{o}(\delta) \to \eta)),$ $\alpha = \mathbf{br}(\mathbf{o}(\gamma \times \delta) \to \eta)$
Recursive tuple match	$\dfrac{\Xi \Vert \Gamma \mid \Delta \vdash \overline{t}^{r} : \beta}{\Xi \Vert \Gamma \mid \Delta \vdash \overline{\Diamond . t}^{r} : \alpha}\ \beta, \gamma, \delta, \eta, \zeta \rhd$	$\beta = \mathbf{br}(\mathbf{r}(\zeta, \gamma) \to \mathbf{br}(\mathbf{r}(\zeta, \delta) \to \eta)),$ $\alpha = \mathbf{br}(\mathbf{r}(\zeta, \gamma \times \delta) \to \eta)$
Player tuple match	$\dfrac{\Xi \Vert \Gamma \mid \Delta \vdash t : \beta}{\Xi \Vert \Gamma \mid \Delta \vdash \Diamond . t : \alpha}\ \beta, \gamma, \delta, \eta \rhd$	$\beta = \mathbf{br}(\gamma \to \mathbf{br}(\delta \to \eta)),$ $\alpha = \mathbf{br}(\gamma \times \delta \to \eta)$
Opponent unit match	$\dfrac{\Xi \Vert \Gamma \mid \Delta \vdash \overline{t}^{o} : \beta}{\Xi \Vert \Gamma \mid \Delta \vdash \overline{\nabla . t}^{o} : \alpha}\ \beta, \gamma \rhd$	$\beta = \mathbf{br}(\gamma), \alpha = \mathbf{br}(\mathbf{o}(()) \to \gamma)$
Recursive unit match	$\dfrac{\Xi \Vert \Gamma \mid \Delta \vdash \overline{t}^{r} : \beta}{\Xi \Vert \Gamma \mid \Delta \vdash \overline{\nabla . t}^{r} : \alpha}\ \beta, \gamma, \delta \rhd$	$\beta = \mathbf{br}(\gamma), \alpha = \mathbf{br}(\mathbf{r}(\delta, ()) \to \gamma)$
Player unit match	$\dfrac{\Xi \Vert \Gamma \mid \Delta \vdash t : \beta}{\Xi \Vert \Gamma \mid \Delta \vdash \overline{\nabla . t} : \alpha}\ \beta, \gamma \rhd$	$\beta = \mathbf{br}(\gamma), \alpha = \mathbf{br}(() \to \gamma)$

Figure 3.36: Rules of type inference for tuples.

$$\text{Fold}\quad \frac{}{\Xi\|\Gamma \mid \Delta \vdash \mathbf{fold} : \alpha}\ \beta, \gamma \rhd \alpha = \forall X.(\mathbf{r}(X, \beta) \to \beta \to \gamma) \to \mathbf{o}(\beta) \to \gamma$$

$$\text{Fold case}\quad \frac{\Xi\|\Gamma \mid \Delta \vdash s : \beta \quad \Xi\|\Gamma \mid \Delta_2 \vdash \overline{t}^r : \gamma}{\Xi\|\Gamma \mid \Delta_1, \Delta_2 \vdash \mathbf{fcase}\ s\ t : \alpha}\ \beta, \gamma, \delta \rhd \gamma = \mathbf{br}(\mathbf{r}(\delta, \beta) \to \alpha)$$

Figure 3.37: Rules of type inference for **fold** constructs.

the inference given in figure 3.43 is as follows:

$$\begin{aligned}
&\exists \alpha_1, \alpha_2.\alpha_0 = \mathbf{o}(\alpha_1) \to \alpha_2, \exists \alpha_3, \alpha_4.\alpha_3 = \mathbf{o}(\alpha_4) \to \alpha_2, \exists \alpha_5.\alpha_4 = \alpha_5 \to \alpha_3,\\
&\quad \exists \alpha_6, \alpha_7.\alpha_4 = \forall X.(\mathbf{r}(X, \alpha_6) \to \alpha_7) \to \mathbf{o}(\alpha_6) \to \alpha_7,\\
&\quad \exists \alpha_8, \alpha_9, \alpha_{10}, \alpha_5 = \mathbf{r}(\alpha_{10}, \alpha_9) \to \alpha_8 \to \alpha_9, \exists \alpha_{11}, \alpha_{12}.\alpha_{10} = \alpha_{11} \to \alpha_{12},\\
&\quad \exists \alpha_{13}, \alpha_{14}, \alpha_{15}.\alpha_{14} = \mathbf{br}(\mathbf{r}(\alpha_{15}, \alpha_{13}) \to \alpha_{12}), \alpha_{13} = \alpha_{11},\\
&\quad \exists \alpha_{16}, \alpha_{17}.\alpha_{14} = \mathbf{br}(\alpha_{16} \to \alpha_{17}), \exists \alpha_{18}, \alpha_{19}, \alpha_{20}, \alpha_{21} \to \alpha_{18} = \mathbf{br}(\mathbf{r}(\alpha_{21}, \alpha_{19}) \to \alpha_{20}),\\
&\quad \alpha_{14} = \mathbf{br}(\mathbf{r}(\alpha_{21}, \mathsf{Nat}) \to \alpha_{20}), \exists \alpha_{22}, \alpha_{23}, \alpha_{24}.\alpha_{22} = \mathbf{br}(\alpha_{23}), \alpha_{18} = \mathbf{br}(\mathbf{r}(\alpha_{24}, ()) \to \alpha_{23}),\\
&\quad \alpha_{22} = \mathsf{Bool}, \exists \alpha_{25}, \alpha_{26}, \alpha_{27}, \alpha_{28}.\alpha_{25} = \mathbf{br}(\mathbf{r}(\alpha_{28}, \alpha_{26}) \to \alpha_{27}),\\
&\quad \alpha_{14} = \mathbf{br}(\mathbf{r}(\alpha_{28}, \mathsf{Nat}) \to \alpha_{27},\\
&\quad \exists \alpha_{29}, \alpha_{30}, \alpha_{31}, \alpha_{32}, \alpha_{33}.\alpha_{29} = \mathbf{br}(\mathbf{r}(\alpha_{33}, \alpha_{30}) \to \mathbf{br}(\mathbf{r}(\alpha_{33}, \alpha_{31}) \to \alpha_{32}),\\
&\quad \alpha_{25} = \mathbf{br}(\mathbf{r}(\alpha_3 3, \alpha_{30} \times \alpha_{31}) \to \alpha_{32}), \exists \alpha_{34}, \alpha_{35}, \alpha_{36}.\alpha_{29} = \mathbf{r}(\alpha_{35}, \alpha_{34}) \to \alpha_{36},\\
&\quad \exists \alpha_{37}, \alpha_{38}, \alpha_{39}.\alpha_{37} = \mathbf{br}(\alpha_{38}), \alpha_{36} = \mathbf{br}(\mathbf{r}(\alpha_{39}, ()) \to \alpha_{38}), \exists \alpha_{40}, \alpha_{41}.\alpha_{37} = \mathbf{br}(\mathbf{r}(\alpha_{41}, \alpha_{40})),\\
&\quad \exists \alpha_{42}, \alpha_{43}.\alpha_{42} = \alpha_{43} \to \alpha_{40}, \alpha_{42} = \mathsf{Bool} \to \mathsf{Bool}, \exists \alpha_{44}, \alpha_{45}.\alpha_{44} = \alpha_{45} \to \alpha_{43},\\
&\quad \alpha_{44} = \alpha_8 \to \alpha_9, \alpha_4 = \alpha_1
\end{aligned}$$

This equation is unified as described by the process in section 3.6.4 to ensure proper typing and to determine the types of symbols.

$$\dfrac{\dfrac{}{\begin{array}{c}(not : \mathsf{Bool} \to \mathsf{Bool}),\\ (f : \alpha_8 \to \alpha_9)\end{array} \| \cdots \mid \vdash f : \alpha_{44}} \rhd \alpha_{44} = \alpha_8 \to \alpha_9}{(not : \mathsf{Bool} \to \mathsf{Bool}), (f : \alpha_8 \to \alpha_9) \| (z : \alpha_1), (x_1 : \alpha_{34}) \mid (x_2 : \alpha_{35}) \vdash f\ x_2 : \alpha_{43}} \alpha_{44}, \alpha_{45} \rhd \alpha_{44} = \alpha_{45} \to \alpha_{43}$$

Figure 3.38: A continuation of the below derivation.

$$\dfrac{}{(not : \mathsf{Bool} \to \mathsf{Bool}),\ (f : \alpha_8 \to \alpha_9) \ \|\cdots|\vdash not : \alpha_{42}} \ \triangleright \alpha_{42} = \mathsf{Bool} \to \mathsf{Bool}$$

Continued in figure 3.38

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(not : \mathsf{Bool} \to \mathsf{Bool}),\ (f : \alpha_8 \to \alpha_9) \ \|(z : \alpha_1), (x_1 : \alpha_{34}) \mid (x_2 : \alpha_{35}) \vdash not\ (f\ x_2) : \alpha_{40}} \ \alpha_{42}, \alpha_{43} \rhd \alpha_{42} = \alpha_{43} \to \alpha_{40}}{(not : \mathsf{Bool} \to \mathsf{Bool}),\ (f : \alpha_8 \to \alpha_9) \ \|(z : \alpha_1), (x_1 : \alpha_{34}) \mid (x_2 : \alpha_{35}) \vdash \overline{not\ (f\ x_2)}^{r} : \alpha_{37}} \ \alpha_{40}, \alpha_{41} \rhd \alpha_{37} = \mathbf{br}(\mathbf{r}(\alpha_{41}, \alpha_{40}))}{(not : \mathsf{Bool} \to \mathsf{Bool}),\ (f : \alpha_8 \to \alpha_9) \ \|(z : \alpha_1), (x_1 : \alpha_{34}) \mid (x_2 : \alpha_{35}) \vdash \overline{\nabla.not\ (f\ x_2)}^{r} : \alpha_{36}} \ \alpha_{37}, \alpha_{38}, \alpha_{39} \rhd \begin{array}{c} \alpha_{37} = \mathbf{br}(\alpha_{38}), \\ \alpha_{36} = \mathbf{br}(\mathbf{r}(\alpha_{39}, ()) \to \alpha_{38}) \end{array}$$

$$\dfrac{\dfrac{\dfrac{}{(not : \mathsf{Bool} \to \mathsf{Bool}),\ (f : \alpha_8 \to \alpha_9) \ \|(z : \alpha_1) \mid \vdash \overline{\lambda^{op} x_1\, x_2.\nabla.\ not\ (f\ x_2)}^{r} : \alpha_{29}} \ \begin{array}{c} \alpha_{29}, \alpha_{30}, \alpha_{31}, \\ \alpha_{32}, \alpha_{33} \end{array} \rhd \begin{array}{c} \alpha_{29} = \mathbf{br}(\mathbf{r}(\alpha_{33}, \alpha_{30}) \to \\ \mathbf{br}(\mathbf{r}(\alpha_{33}, \alpha_{31}) \to \alpha_{32}), \\ \alpha_{25} = \mathbf{br}(\mathbf{r}(\alpha_{33}, \alpha_{30} \times \alpha_{31}) \to \alpha_{32}) \end{array}}{(not : \mathsf{Bool} \to \mathsf{Bool}),\ (f : \alpha_8 \to \alpha_9) \ \|(z : \alpha_1) \mid \vdash \overline{\Diamond.\lambda^{op} x_1\, x_2.\nabla.\ not\ (f\ x_2)}^{r} : \alpha_{25}} \ \begin{array}{c} \alpha_{25}, \alpha_{26}, \\ \alpha_{27}, \alpha_{28} \end{array} \rhd \begin{array}{c} \alpha_{25} = \mathbf{br}(\mathbf{r}(\alpha_{28}, \alpha_{26}) \to \alpha_{27}), \\ \alpha_{14} = \mathbf{br}(\mathbf{r}(\alpha_{28}, \mathsf{Nat})) \to \alpha_{27}) \end{array}$$

$$(not : \mathsf{Bool} \to \mathsf{Bool}),\ (f : \alpha_8 \to \alpha_9) \ \|(z : \alpha_1) \mid \vdash \overline{\lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_1\, x_2.\nabla.\ not\ (f\ x_2)}^{r} : \alpha_{14}$$

Figure 3.39: A continuation of the below derivation.

$$\dfrac{\dfrac{\dfrac{\vdots}{\cdots \| (z : \alpha_1) \mid \vdash \mathsf{True}\ () : \alpha_{22}}\ \rhd \alpha_{22} = \mathsf{Bool}}{\cdots \| (z : \alpha_1) \mid \vdash \overline{\mathsf{True}\ ()}^{r} : \alpha_{22}}}{\cdots \| (z : \alpha_1) \mid \vdash \overline{\forall.\mathsf{True}\ ()}^{r} : \alpha_{18}}\ \begin{array}{c}\alpha_{22},\\ \alpha_{23}, \alpha_{24}\end{array}\ \rhd\ \begin{array}{c}\alpha_{22} = \mathbf{br}(\alpha_{23}),\\ \alpha_{18} = \mathbf{br}(\mathbf{r}(\alpha_{24}, ()) \to \alpha_{23})\end{array}$$

$$\dfrac{\cdots \| (z : \alpha_1) \mid \vdash \overline{\forall.\mathsf{True}\ ()}^{r} : \alpha_{18}}{\cdots \| (z : \alpha_1) \mid \vdash \overline{\lambda^{\mathsf{C}}\mathsf{Zero}.\forall.\mathsf{True}\ ()}^{r} : \alpha_{14}}\ \begin{array}{c}\alpha_{18},\\ \alpha_{19},\\ \alpha_{20},\\ \alpha_{21}\end{array}\ \rhd\ \begin{array}{c}\alpha_{18} = \mathbf{br}(\mathbf{r}(\alpha_{21}, \alpha_{19}) \to \alpha_{20}),\\ \alpha_{14} = \mathbf{br}(\mathbf{r}(\alpha_{21}, \mathsf{Nat}) \to \alpha_{20})\end{array}$$

Continued in figure 3.39

$$\dfrac{}{\begin{array}{c}(not : \mathsf{Bool} \to \mathsf{Bool}),\\ (f : \alpha_8 \to \alpha_9)\end{array} \| (z : \alpha_1) \mid \vdash \overline{\lambda^{\mathsf{C}}\mathsf{Zero}.\forall.\mathsf{True}\ () +\ \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_1\, x_2.\forall.\ not\ (f\ x_2)}^{r} : \alpha_{14}}\ \alpha_{16}, \alpha_{17} \rhd \alpha_{14} = \mathbf{br}(\alpha_{16} \to \alpha_{17})$$

Figure 3.40: A continuation of the below derivation.

$$\dfrac{\dfrac{\dfrac{}{\cdots \| (z : \alpha_1) \mid (x : \alpha_{11}) \vdash x : \alpha_{13}}\ \rhd \alpha_{13} = \alpha_{11} \quad \text{Continued in figure 3.40}}{(not : \mathsf{Bool} \to \mathsf{Bool}),\ (f : \alpha_8 \to \alpha_9) \quad \| (z : \alpha_1) \mid (x : \alpha_{11}) \vdash \begin{array}{c}\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{True}\ () \\ +\ \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_1\, x_2.\triangledown. \\ not\ (f\ x_2))\end{array} : \alpha_{12}}\ \alpha_{13}, \alpha_{14}, \alpha_{15} \rhd \alpha_{14} = \mathbf{br}(\mathbf{r}(\alpha_{15}, \alpha_{13}) \to \alpha_{12})}{(not : \mathsf{Bool} \to \mathsf{Bool}),\ (f : \alpha_8 \to \alpha_9) \quad \| (z : \alpha_1) \mid \vdash \begin{array}{c}\lambda^p x.\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{True}\ () \\ +\ \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_1\, x_2.\triangledown.(not\ (f\ x_2)))\end{array} : \alpha_{10}}\ \alpha_{11}, \alpha_{12} \rhd \alpha_{10} = \alpha_{11} \to \alpha_{12}}{(not : \mathsf{Bool} \to \mathsf{Bool}) \| (z : \alpha_1) \mid \vdash \begin{array}{c}\lambda^f f.\lambda^p x.\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{True}\ () \\ +\ \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_1\, x_2.\triangledown.(not\ (f\ x_2)))\end{array} : \alpha_5}\ \alpha_8, \alpha_9, \alpha_{10} \rhd \alpha_5 = \mathbf{r}(\alpha_{10}, \alpha_9) \to \alpha_8 \to \alpha_9$$

Figure 3.41: A continuation of the below derivation.

$$\dfrac{\dfrac{}{\cdots \| (z : \alpha_1) \mid \vdash \mathbf{fold} : \alpha_4}\ \alpha_6, \alpha_7 \rhd \alpha_4 = \forall X.(\mathbf{r}(X, \alpha_6) \to \alpha_7) \to \mathbf{o}(\alpha_6) \to \alpha_7 \quad \text{Continued in figure 3.41}}{(not : \mathsf{Bool} \to \mathsf{Bool}) \| (z : \alpha_1) \mid \vdash \begin{array}{c}\mathbf{fold}\ (\lambda^f f.\lambda^p x.\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{True}\ () \\ +\ \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_1\, x_2.\triangledown.(not\ (f\ x_2))))\end{array} : \alpha_3}\ \alpha_4, \alpha_5 \rhd \alpha_4 = \alpha_5 \to \alpha_3$$

Figure 3.42: A continuation of the below derivation.

$$\dfrac{\text{Continued in figure 3.42} \quad \dfrac{}{(not : \mathsf{Bool} \to \mathsf{Bool})\|(z : \alpha_1) \mid \vdash z : \alpha_4} \rhd \alpha_4 = \alpha_1}{\begin{array}{c}(not : \mathsf{Bool} \to \mathsf{Bool})\|(z : \alpha_1) \mid \vdash \ (\mathbf{fold}\ (\lambda^f f.\lambda^p x.\mathbf{fcase}\ x \\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{True}\ () + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_1\, x_2.\triangledown.(not\ (f\ x_2))))))\ : \alpha_2 \\ (z)_o\end{array}} \alpha_3, \alpha_4 \rhd \alpha_3 = \mathbf{o}(\alpha_4) \to \alpha_2$$

$$\dfrac{\begin{array}{c}(not : \mathsf{Bool} \to \mathsf{Bool})\|(z : \alpha_1) \mid \vdash \ (\mathbf{fold}\ (\lambda^f f.\lambda^p x.\mathbf{fcase}\ x \\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{True}\ () + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_1\, x_2.\triangledown.(not\ (f\ x_2))))))\ : \alpha_2 \\ (z)_o\end{array}}{(not : \mathsf{Bool} \to \mathsf{Bool})\| \ \mid \vdash \left(\begin{array}{c}\lambda^o z.(\mathbf{fold}\ (\lambda^f f.\lambda^p x.\mathbf{fcase}\ x \\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{True}\ () + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_1\, x_2.\triangledown.(not\ (f\ x_2))))) \\ (z)_o\end{array}\right) : \alpha_0} \alpha_1, \alpha_2 \rhd \alpha_0 = \mathbf{o}(\alpha_1) \to \alpha_2$$

Figure 3.43: Type inference of the *parity* function given in figure 3.20.

$$\text{Non-recursive co-branch} \quad \frac{(\mathsf{D}_i : \gamma \to \delta) \in \Xi \quad \Xi \| \Gamma \mid \Delta \vdash t : \beta}{\Xi \| \Gamma \mid \Delta \vdash \lambda^{\mathsf{D}} \mathsf{D}_i.t : \alpha} \; \beta \rhd \beta = \delta, \alpha = \bar{\mathbf{br}}(\gamma)$$

$$\text{Co-branch composition} \quad \frac{\Xi \| \Gamma \mid \Delta \vdash t : \beta \quad \Xi \| \Gamma \mid \Delta \vdash u : \gamma}{\Xi \| \Gamma \mid \Delta \vdash t \oplus u : \alpha} \; \beta, \gamma, \delta \rhd \alpha = \beta, \alpha = \gamma, \alpha = \bar{\mathbf{br}}(\delta)$$

$$\text{Record} \quad \frac{}{\Xi \| \Gamma \mid \Delta \vdash \mathbf{record} : \alpha} \; \beta \rhd \alpha = \bar{\mathbf{br}}(\beta)$$

Figure 3.44: Rules of inference for non-recursive coinductive terms.

3.6.3 Coinductive terms

Figure 3.44 gives the type inference rules for non-recursive coinductive terms. These correspond directly to rules for non-recursive coinductive terms given in the type checking section in figure 3.25.

Figure 3.45 gives the type inference rules for recursive coinductive terms. Only the rules for nullary and unary **unfold** are given, though rules exist for every arity. For **unfold** constructs where the g function has arity greater than 1, the type equations for α and δ are modified as appropriate, and the extra parameters are introduced into the player context along with x. Note that the arity needs to be known at the time of inference and hence the structure of an **unfold** is strict.

Figure 3.48 shows the type inference process on the *allNats* function given in figure 3.26. After unification, the equations generated would correctly infer a type of InfList(Nat) for the *allNats* function.

Non-recursive co-branch of recursive record

$$\frac{(\mathsf{D}_i : \gamma \to \delta) \in \Xi,\ \gamma \neq \delta \qquad \Xi \| \Gamma \mid \Delta \vdash t : \beta}{\Xi \| \Gamma \mid \Delta \vdash \overline{\lambda^{\mathsf{D}} \mathsf{D}_i . t}^{r} : \alpha} \ \beta, \eta \rhd \ \begin{array}{c} \beta = \delta, \\ \alpha = \bar{\mathbf{br}}(\mathbf{r}(\eta, \gamma)) \end{array}$$

Recursive co-branch

$$\frac{(\mathsf{D}_i : \gamma \to \gamma) \in \Xi \quad \Xi \| \Gamma \mid \Delta \vdash t : \beta}{\Xi \| \Gamma \mid \Delta \vdash \overline{\lambda^{\mathsf{D}} \mathsf{D}_i . t}^{r} : \alpha} \ \beta, \rhd\alpha = \bar{\mathbf{br}}(\mathbf{r}(\beta, \gamma))$$

Composition of recursive co-banches

$$\frac{\Xi \| \Gamma \mid \Delta \vdash \overline{t}^{r} : \alpha \quad \Xi \| \Gamma \mid \Delta \vdash \overline{u}^{r} : \alpha}{\Xi \| \Gamma \mid \Delta \vdash \overline{t \oplus u}^{r} : \alpha}$$

Nullary unfold

$$\frac{(g : () \to X), \Xi \| \Gamma \mid \ \vdash \overline{t}^{r} : \gamma}{\Xi \| \Gamma \mid \Delta \vdash \mathbf{unfold}\ (\lambda^{\bar{f}} g . t) : \alpha} \ \beta \rhd \forall X . \beta = \bar{\mathbf{br}}(\mathbf{r}(X, \alpha))$$

Unary unfold

$$\frac{(g : \beta \to X), \Xi \| \Gamma \mid (x : \beta) \vdash \overline{t}^{r} : \gamma}{\Xi \| \Gamma \mid \Delta \vdash \mathbf{unfold}\ (\lambda^{\bar{f}} g . \lambda^{p} x . t) : \alpha} \ \begin{array}{c} \beta, \gamma, \\ \delta \end{array} \ \rhd \ \begin{array}{c} \forall X . \gamma = \bar{\mathbf{br}}(\mathbf{r}(X, \delta)), \\ \alpha = \mathbf{o}(\beta) \to \delta \end{array}$$

Figure 3.45: Rules of inference for recursive coinductive terms.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(g:\alpha_3\to X)\|\ \mid (x:\alpha_3)\vdash g:\alpha_{11}}\ \triangleright\alpha_{11}=\alpha_3\to X \quad \dfrac{\dfrac{\dfrac{}{\cdots\|\ \mid (x:\alpha_3)\vdash x:\alpha_{14}}\ \triangleright\alpha_{14}=\alpha_3 \quad \dfrac{}{\cdots\|\ \mid \cdots\vdash():()}\ \triangleright\alpha_{15}=()}{(g:\alpha_3\to X)\|\ \mid (x:\alpha_3)\vdash x\times():\alpha_{13}}\ \alpha_{14},\alpha_{15}\triangleright\alpha_{13}=\alpha_{14}\times\alpha_{15}}{(g:\alpha_3\to X)\|\ \mid (x:\alpha_3)\vdash \mathsf{Succ}\ (x\times()):\alpha_{12}}\ \alpha_{13}\triangleright\alpha_{12}=\mathsf{Nat},\alpha_{13}=(\mathsf{Nat}\times())}{(g:\alpha_3\to X)\|\ \mid (x:\alpha_3)\vdash g\ (\mathsf{Succ}\ (x\times())):\alpha_9}\ \alpha_{11},\alpha_{12}\triangleright\alpha_{11}=\alpha_{12}\to\alpha_9}{(g:\alpha_3\to X)\|\ \mid (x:\alpha_3)\vdash \lambda^{\mathsf{D}}\mathsf{Tail}.g\ (\mathsf{Succ}\ (x\times())):\alpha_4}\ \alpha_9,\alpha_{10}\triangleright\alpha_4=\bar{\mathbf{br}}(\mathbf{r}(\alpha_9,\mathsf{InfList}(\alpha_{10})))}{(g:\alpha_3\to X)\|\ \mid (x:\alpha_3)\vdash \overline{\lambda^{\mathsf{D}}\mathsf{Tail}.g\ (\mathsf{Succ}\ (x\times()))}^{r}:\alpha_4}$$

Figure 3.46: A continuation of the derivation below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(g:\alpha_3\to X)\|\ \mid (x:\alpha_3)\vdash x:\alpha_7}\ \triangleright\alpha_7=\alpha_3}{(g:\alpha_3\to X)\|\ \mid (x:\alpha_3)\vdash \overline{x}^{r}:\alpha_7}}{(g:\alpha_3\to X)\|\ \mid (x:\alpha_3)\vdash \lambda^{\mathsf{D}}\mathsf{Head}.x:\alpha_4}\ \alpha_7,\alpha_8\triangleright\alpha_4=\bar{\mathbf{br}}(\mathbf{r}(\alpha_8,\mathsf{InfList}(\alpha_7))) \quad \text{Continued in figure 3.46}}{(g:\alpha_3\to X)\|\ \mid (x:\alpha_3)\vdash \overline{\lambda^{\mathsf{D}}\mathsf{Head}.x\oplus\lambda^{\mathsf{D}}\mathsf{Tail}.g\ (\mathsf{Succ}\ (x\times()))}^{r}:\alpha_4}\ \begin{matrix}\alpha_3,\alpha_4,\\ \alpha_5\end{matrix}\ \triangleright\ \begin{matrix}\forall X.\alpha_4=\bar{\mathbf{br}}(\mathbf{r}(X,\alpha_5)),\\ \alpha_1=\mathbf{o}(\alpha_3)\to\alpha_5\end{matrix}}{\|\ \mid\ \vdash \mathbf{unfold}\ (\lambda^{\bar{f}}g.\lambda^{p}x.(\lambda^{\mathsf{D}}\mathsf{Head}.x\oplus\lambda^{\mathsf{D}}\mathsf{Tail}.g\ (\mathsf{Succ}\ (x\times()))))):\alpha_1}$$

Figure 3.47: A continuation of the derivation below.

$$\dfrac{\text{Continued in figure 3.47} \quad \dfrac{\vdots}{\|\ \mid\ \vdash \mathsf{Zero}\ ():\alpha_2}\ \triangleright\alpha_2=\mathsf{Nat}}{\|\ \mid\ \vdash \mathbf{unfold}\ (\lambda^{\bar{f}}g.\lambda^{p}x.(\lambda^{\mathsf{D}}\mathsf{Head}.x\oplus\lambda^{\mathsf{D}}\mathsf{Tail}.g\ (\mathsf{Succ}\ (x\times()))))\ (\mathsf{Zero}\ ())_{\mathbf{o}}:\alpha_0}\ \alpha_1,\alpha_2\triangleright\alpha_1=\mathbf{o}(\alpha_2)\to\alpha_0$$

Figure 3.48: Type inference on the *allNats* function given in figure 3.26.

3.6.4 Unification

From the process of type inference we collect a set of type equations. Consider a sequent of the following form:

$$\frac{s_2 \quad s_3}{s_1}\ \beta_1, \ldots, \beta_m \rhd \gamma_1 = \delta_1, \ldots, \gamma_n = \delta_n$$

Consider further that s_2 generates type equations Φ_2 and that s_3 generates type equations Φ_3. The type equations created by the given sequent are defined to be $\Phi_1 = (\exists\beta_1.\cdots.\exists\beta_m.\gamma_1 = \delta_1, \ldots, \gamma_1 = \delta_n, \Phi_2, \Phi_3)$.

More precisely, a set of type equations is defined to be:

Empty set	$\Phi := \emptyset$
Equation	$\Phi := \alpha = \delta$
List of equations	$\Phi := \Phi, \Phi$
Existentially quantified type variable	$\Phi := \exists\alpha.\Phi$
Universally quantified type variable	$\Phi := \forall\alpha.\Phi$

By the process of unification we desire both to ensure that a given set of type equations is satisfiable and to determine the most general types for the type variables which would make the type equations satisfiable.

The algorithm for type equation unification in Pola is Hindley-Milner type inference [9, 28, 29] with the complication of ensuring that universal types—those introduced by **fold** and **unfold** constructs to maintain polynomial time constraints—are enforced.

The algorithm removes equations one at a time, starting at the rightmost equation. Each equation removed necessarily adds a binding with a new type variable. Each time a binding is added, we unify that binding against existing bindings and against type equations yet to be removed. When the type equations yet to be removed are unified against a new binding, they may generate new equations. The notation $\Phi[\delta/\alpha]$ refers to Φ with the variable δ substituted for the type equation α.

Where equations are to the left of the $\|$ symbol and bindings are to the right of the $\|$ symbol, we describe the unification process via the following rewriting rules. δ does not contain the variable α, Υ does not contain the variable α, and β is a type variable such that $\beta \neq \alpha$.

$$\Phi, \exists\alpha.\delta \parallel \Psi \Rightarrow \Phi, \delta \parallel \Psi \tag{3.1}$$

$$\Phi, \alpha = \delta \parallel \Psi \Rightarrow \Phi[\delta/\alpha] \parallel \Psi[\delta/\alpha], \alpha = \delta \tag{3.2}$$

$$\Phi, \forall\alpha.\emptyset \parallel \alpha = \beta, \Psi \Rightarrow \Phi \parallel \Psi \tag{3.3}$$

$$\Phi, \forall\alpha.\emptyset \parallel \Upsilon \Rightarrow \Phi \parallel \Upsilon \tag{3.4}$$

Note, in particular, that there is no rewriting rule to remove a universal quantifier, $\forall\alpha$, in the case that α is bound to a concrete type. This would indicate a typing error.

Unification stops when all equations have been removed, which indicates successful typing, or when there is no rule to apply, which indicates failed typing.

Definition 1. *A type equation,* Φ*,* unifies successfully*, denoted* $\Phi \rightsquigarrow \Phi'$*, if and only if unification of* Φ *reaches* $\emptyset$

Definition 2. *A type equation Φ generates a binding, α', for a type variable α, denoted $\Phi_\alpha \rightsquigarrow \alpha'$, if and only if one of the following two conditions is true:*

1. $\Phi \rightsquigarrow \Phi'$ *and* $(\alpha = \alpha') \in \Phi'$*; or*

2. $\Phi \rightsquigarrow \Phi'$ *and* $\alpha = \alpha'$ *and* $\nexists\beta.(\alpha = \beta) \in \Phi'$.

Substitution and matching

Unifying involves substituting type variables. $\Phi[\delta/\alpha]$ is defined as the type equations Φ where the type δ is replaced by the type variable α. This involves matching type equations in α with the given substitution and potentially creating more type equations. For instance, matching the types $\mathsf{List}(\alpha_3)$ and $\mathsf{List}(\mathsf{Nat})$ generates the equation $\alpha_3 = \mathsf{Nat}$ where α_3 is a type variable.

First we must consider when two types are considered to match. Much of the process of unification is determining whether type equations generated in disparate parts of the derivation tree of type inference match and, if they do match, which new type equations are generated. If a program is not well-typed, it is only because some type equations will be found to not match.

Definition 3. *We say $\mathbf{v}(\alpha)$ to denote that the type α is a type variable.*

Definition 4. *We say $\alpha \hookleftarrow \beta$ if and only if the type α contains the type variable β. $\alpha \hookleftarrow \beta$ if and only if at least one of the following is true:*

- $\alpha = \beta$*;*

- $\alpha = \mathcal{T}_x(\alpha_1, \ldots, \alpha_n)$ *for some structure x, using the type structure notation given in section 3.2.1, and types $\alpha_1, \ldots, \alpha_n$ and there is an i such that $1 \leq i \leq n$ such that $\alpha_i \hookleftarrow \beta$.*

The rules of matching, denoted $\alpha \stackrel{\circ}{=} \beta$ (α matches with β, which may generate new type equations, are as follows:

$$\alpha \stackrel{\circ}{=} \alpha \Rightarrow \emptyset \tag{3.5}$$

$$\alpha \stackrel{\circ}{=} \beta \Rightarrow \alpha = \beta \qquad \text{where } \mathbf{v}(\alpha) \text{ and } \beta \not\hookleftarrow \alpha \tag{3.6}$$

$$\alpha \stackrel{\circ}{=} \beta \Rightarrow \beta = \alpha \qquad \text{where } \mathbf{v}(\beta) \text{ and } \alpha \not\hookleftarrow \beta \tag{3.7}$$

$$\mathcal{T}_x(\alpha_1, \ldots, \alpha_n) \stackrel{\circ}{=} \mathcal{T}_x(\beta_1, \ldots, \beta_n) \Rightarrow \alpha_1 \stackrel{\circ}{=} \beta_1, \ldots, \alpha_n \stackrel{\circ}{=} \beta_n \tag{3.8}$$

All other cases are a mismatch error and indicate a Compositional Pola program which is not well-typed.

Now that we have a definition of two types matching, we can define how substitution of types is performed in type equations. The rules of substitution are as follows, where

$\gamma \neq \alpha$. In each of these cases, the type δ is substituted for the type variable α:

$$(\emptyset)[\delta/\alpha] \Rightarrow \emptyset \tag{3.9}$$
$$(\alpha = \delta)[\delta/\alpha] \Rightarrow \emptyset \tag{3.10}$$
$$(\alpha = \gamma)[\delta/\alpha] \Rightarrow \gamma \stackrel{\circ}{=} \delta \tag{3.11}$$
$$(\Phi, \Upsilon)[\delta/\alpha] \Rightarrow \Phi[\delta/\alpha], \Upsilon[\delta/\alpha] \tag{3.12}$$
$$(\exists\gamma.\Phi)[\delta/\alpha] \Rightarrow \exists\gamma.(\Phi[\delta/\alpha]) \tag{3.13}$$
$$(\forall\gamma.\Phi)[\delta/\alpha] \Rightarrow \forall\gamma.(\Phi[\delta/\alpha]) \tag{3.14}$$

The following shows the process of unification of the type equation generated from the allNats function given in section 3.6.3:

$$\begin{array}{l}\exists\alpha_0.\exists\alpha_1.\exists\alpha_2.\alpha_1 = (\mathbf{o}(\alpha_2) \to \alpha_0), \alpha_2 = \mathsf{Nat}, \exists\alpha_3.\exists\alpha_4.\exists\alpha_5.\forall X.\alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(X, \alpha_5)),\\ \quad \alpha_1 = (\mathbf{o}(\alpha_3) \to \alpha_5), \exists\alpha_7.\exists\alpha_8.\alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(\alpha_8, \mathsf{InfList}(\alpha_7))), \alpha_7 = \alpha_3,\\ \quad \exists\alpha_9.\exists\alpha_{10}.\alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(\alpha_9, \mathsf{InfList}(\alpha_{10}))), \exists\alpha_{11}.\exists\alpha_{12}.\alpha_{11} = (\alpha_{12} \to \alpha_9), \alpha_{11} = (\alpha_3 \to X),\\ \quad \exists\alpha_{13}.\alpha_{12} = \mathsf{Nat}, \alpha_{13} = (\mathsf{Nat} \times ()), \exists\alpha_{14}.\exists\alpha_{15}.\alpha_{13} = \alpha_{14} \times \alpha_{15}, \alpha_{14} = \alpha_3, \alpha_{15} = ()\end{array}$$

$$\begin{array}{l}\alpha_1 = (\mathbf{o}(\alpha_2) \to \alpha_0), \alpha_2 = \mathsf{Nat}, \forall X.\alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(X, \alpha_5)), \alpha_1 = (\mathbf{o}(\alpha_3) \to \alpha_5),\\ \quad \alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(\alpha_8, \mathsf{InfList}(\alpha_7))), \alpha_7 = \alpha_3, \alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(\alpha_9, \mathsf{InfList}(\alpha_{10}))), \alpha_{11} = (\alpha_{12} \to \alpha_9),\\ \quad \alpha_{11} = (\alpha_3 \to X), \alpha_{12} = \mathsf{Nat}, \alpha_{13} = (\mathsf{Nat} \times ()), \alpha_{13} = \alpha_{14} \times \alpha_{15}, \alpha_{14} = \alpha_3, \alpha_{15} = ()\end{array}$$

$$\left.\begin{array}{l}\alpha_1 = (\mathbf{o}(\alpha_2) \to \alpha_0), \alpha_2 = \mathsf{Nat}, \forall X.\alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(X, \alpha_5)),\\ \quad \alpha_1 = (\mathbf{o}(\alpha_3) \to \alpha_5), \alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(\alpha_8, \mathsf{InfList}(\alpha_7))), \alpha_7 = \alpha_3,\\ \quad \alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(\alpha_9, \mathsf{InfList}(\alpha_{10}))), \alpha_{11} = (\alpha_{12} \to \alpha_9), \alpha_{11} = (\alpha_3 \to X),\\ \quad \alpha_{12} = \mathsf{Nat}, \alpha_{13} = (\mathsf{Nat} \times ()), \alpha_{13} = \alpha_{14} \times (), \alpha_{14} = \alpha_3\end{array}\right\| \begin{array}{l}\alpha_{15} = ()\end{array}$$

$$\left.\begin{array}{l}\alpha_1 = (\mathbf{o}(\alpha_2) \to \alpha_0), \alpha_2 = \mathsf{Nat}, \forall X.\alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(X, \alpha_5)),\\ \quad \alpha_1 = (\mathbf{o}(\alpha_3) \to \alpha_5), \alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(\alpha_8, \mathsf{InfList}(\alpha_7))), \alpha_7 = \alpha_3,\\ \quad \alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(\alpha_9, \mathsf{InfList}(\alpha_{10}))), \alpha_{11} = (\alpha_{12} \to \alpha_9), \alpha_{11} = (\alpha_3 \to X),\\ \quad \alpha_{12} = \mathsf{Nat}, \alpha_{13} = (\mathsf{Nat} \times ()), \alpha_{13} = \alpha_3 \times ()\end{array}\right\| \begin{array}{l}\alpha_{14} = \alpha_3\\ \alpha_{15} = ()\end{array}$$

$$\left.\begin{array}{l}\alpha_1 = (\mathbf{o}(\alpha_2) \to \alpha_0), \alpha_2 = \mathsf{Nat}, \forall X.\alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(X, \alpha_5)),\\ \quad \alpha_1 = (\mathbf{o}(\alpha_3) \to \alpha_5), \alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(\alpha_8, \mathsf{InfList}(\alpha_7))), \alpha_7 = \alpha_3,\\ \quad \alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(\alpha_9, \mathsf{InfList}(\alpha_{10}))), \alpha_{11} = (\alpha_{12} \to \alpha_9), \alpha_{11} = (\alpha_3 \to X),\\ \quad \alpha_{12} = \mathsf{Nat}, \alpha_3 = \mathsf{Nat}\end{array}\right\| \begin{array}{l}\alpha_{13} = \alpha_3 \times ()\\ \alpha_{14} = \alpha_3\\ \alpha_{15} = ()\end{array}$$

$$\left.\begin{array}{l}\alpha_1 = (\mathbf{o}(\alpha_2) \to \alpha_0), \alpha_2 = \mathsf{Nat}, \forall X.\alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(X, \alpha_5)),\\ \quad \alpha_1 = (\mathbf{o}(\mathsf{Nat}) \to \alpha_5), \alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(\alpha_8, \mathsf{InfList}(\alpha_7))), \alpha_7 = \mathsf{Nat},\\ \quad \alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(\alpha_9, \mathsf{InfList}(\alpha_{10}))), \alpha_{11} = (\alpha_{12} \to \alpha_9), \alpha_{11} = (\mathsf{Nat} \to X),\\ \quad \alpha_{12} = \mathsf{Nat}\end{array}\right\| \begin{array}{l}\alpha_3 = \mathsf{Nat}\\ \alpha_{13} = \mathsf{Nat} \times ()\\ \alpha_{14} = \mathsf{Nat}\\ \alpha_{15} = ()\end{array}$$

$$\begin{array}{l}\alpha_1 = (\mathbf{o}(\alpha_2) \to \alpha_0), \alpha_2 = \mathsf{Nat}, \forall X.\alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(X, \alpha_5)),\\ \quad \alpha_1 = (\mathbf{o}(\mathsf{Nat}) \to \alpha_5), \alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(\alpha_8, \mathsf{InfList}(\alpha_7))), \alpha_7 = \mathsf{Nat},\\ \quad \alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(\alpha_9, \mathsf{InfList}(\alpha_{10}))), \alpha_{11} = (\mathsf{Nat} \to \alpha_9), \alpha_{11} = (\mathsf{Nat} \to X)\end{array} \left\| \begin{array}{rl} \alpha_3 = & \mathsf{Nat}\\ \alpha_{12} = & \mathsf{Nat}\\ \alpha_{13} = & \mathsf{Nat} \times ()\\ \alpha_{14} = & \mathsf{Nat}\\ \alpha_{15} = & () \end{array} \right.$$

$$\begin{array}{l}\alpha_1 = (\mathbf{o}(\alpha_2) \to \alpha_0), \alpha_2 = \mathsf{Nat}, \forall X.\alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(X, \alpha_5)),\\ \quad \alpha_1 = (\mathbf{o}(\mathsf{Nat}) \to \alpha_5), \alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(\alpha_8, \mathsf{InfList}(\alpha_7))), \alpha_7 = \mathsf{Nat},\\ \quad \alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(\alpha_9, \mathsf{InfList}(\alpha_{10}))), X = \alpha_9\end{array} \left\| \begin{array}{rl} \alpha_3 = & \mathsf{Nat}\\ \alpha_{11} = & \mathsf{Nat} \to X\\ \alpha_{12} = & \mathsf{Nat}\\ \alpha_{13} = & \mathsf{Nat} \times ()\\ \alpha_{14} = & \mathsf{Nat}\\ \alpha_{15} = & () \end{array} \right.$$

$$\begin{array}{l}\alpha_1 = (\mathbf{o}(\alpha_2) \to \alpha_0), \alpha_2 = \mathsf{Nat}, \forall X.\alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(\alpha_9, \alpha_5)),\\ \quad \alpha_1 = (\mathbf{o}(\mathsf{Nat}) \to \alpha_5), \alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(\alpha_8, \mathsf{InfList}(\alpha_7))), \alpha_7 = \mathsf{Nat},\\ \quad \alpha_4 = \bar{\mathbf{br}}(\mathbf{r}(\alpha_9, \mathsf{InfList}(\alpha_{10})))\end{array} \left\| \begin{array}{rl} \alpha_3 = & \mathsf{Nat}\\ \alpha_{11} = & \mathsf{Nat} \to \alpha_9\\ \alpha_{12} = & \mathsf{Nat}\\ \alpha_{13} = & \mathsf{Nat} \times ()\\ \alpha_{14} = & \mathsf{Nat}\\ \alpha_{15} = & ()\\ X = & \alpha_9 \end{array} \right.$$

$$\begin{array}{l}\alpha_1 = (\mathbf{o}(\alpha_2) \to \alpha_0), \alpha_2 = \mathsf{Nat}, \forall X.\alpha_5 = \mathsf{InfList}(\alpha_{10}),\\ \quad \alpha_1 = (\mathbf{o}(\mathsf{Nat}) \to \alpha_5), \alpha_8 = \alpha_9, \alpha_7 = \alpha_{10},\\ \quad \alpha_7 = \mathsf{Nat}\end{array} \left\| \begin{array}{rl} \alpha_3 = & \mathsf{Nat}\\ \alpha_4 = & \bar{\mathbf{br}}(\mathbf{r}(\alpha_9, \mathsf{InfList}(\alpha_{10})))\\ \alpha_{11} = & \mathsf{Nat} \to \alpha_9\\ \alpha_{12} = & \mathsf{Nat}\\ \alpha_{13} = & \mathsf{Nat} \times ()\\ \alpha_{14} = & \mathsf{Nat}\\ \alpha_{15} = & ()\\ X = & \alpha_9 \end{array} \right.$$

$$\begin{array}{l}\alpha_1 = (\mathbf{o}(\alpha_2) \to \alpha_0), \alpha_2 = \mathsf{Nat}, \forall X.\alpha_5 = \mathsf{InfList}(\alpha_{10}),\\ \quad \alpha_1 = (\mathbf{o}(\mathsf{Nat}) \to \alpha_5), \alpha_8 = \alpha_9, \alpha_{10} = \mathsf{Nat}\end{array} \left\| \begin{array}{rl} \alpha_3 = & \mathsf{Nat}\\ \alpha_4 = & \bar{\mathbf{br}}(\mathbf{r}(\alpha_9, \mathsf{InfList}(\alpha_{10})))\\ \alpha_7 = & \mathsf{Nat}\\ \alpha_{11} = & \mathsf{Nat} \to \alpha_9\\ \alpha_{12} = & \mathsf{Nat}\\ \alpha_{13} = & \mathsf{Nat} \times ()\\ \alpha_{14} = & \mathsf{Nat}\\ \alpha_{15} = & ()\\ X = & \alpha_9 \end{array} \right.$$

$$\begin{array}{l} \alpha_1 = (\mathbf{o}(\alpha_2) \to \alpha_0), \alpha_2 = \mathsf{Nat}, \forall X.\alpha_5 = \mathsf{InfList}(\mathsf{Nat}), \\ \quad \alpha_1 = (\mathbf{o}(\mathsf{Nat}) \to \alpha_5), \alpha_8 = \alpha_9 \end{array} \left\| \begin{array}{rl} \alpha_3 = & \mathsf{Nat} \\ \alpha_4 = & \bar{\mathbf{b}}\mathbf{r}(\mathbf{r}(\alpha_9, \mathsf{InfList}(\mathsf{Nat}))) \\ \alpha_7 = & \mathsf{Nat} \\ \alpha_{10} = & \mathsf{Nat} \\ \alpha_{11} = & \mathsf{Nat} \to \alpha_9 \\ \alpha_{12} = & \mathsf{Nat} \\ \alpha_{13} = & \mathsf{Nat} \times () \\ \alpha_{14} = & \mathsf{Nat} \\ \alpha_{15} = & () \\ X = & \alpha_9 \end{array} \right.$$

$$\begin{array}{l} \alpha_1 = (\mathbf{o}(\alpha_2) \to \alpha_0), \alpha_2 = \mathsf{Nat}, \forall X.\alpha_5 = \mathsf{InfList}(\mathsf{Nat}), \\ \quad \alpha_1 = (\mathbf{o}(\mathsf{Nat}) \to \alpha_5) \end{array} \left\| \begin{array}{rl} \alpha_3 = & \mathsf{Nat} \\ \alpha_4 = & \bar{\mathbf{b}}\mathbf{r}(\mathbf{r}(\alpha_9, \mathsf{InfList}(\mathsf{Nat}))) \\ \alpha_7 = & \mathsf{Nat} \\ \alpha_8 = & \alpha_9 \\ \alpha_{10} = & \mathsf{Nat} \\ \alpha_{11} = & \mathsf{Nat} \to \alpha_9 \\ \alpha_{12} = & \mathsf{Nat} \\ \alpha_{13} = & \mathsf{Nat} \times () \\ \alpha_{14} = & \mathsf{Nat} \\ \alpha_{15} = & () \\ X = & \alpha_9 \end{array} \right.$$

$$\alpha_2 = \mathsf{Nat}, \alpha_0 = \alpha_5, \alpha_2 = \mathsf{Nat}, \forall X.\alpha_5 = \mathsf{InfList}(\mathsf{Nat}), \left\| \begin{array}{rl} \alpha_1 = & (\mathbf{o}(\mathsf{Nat}) \to \alpha_5) \\ \alpha_3 = & \mathsf{Nat} \\ \alpha_4 = & \bar{\mathbf{b}}\mathbf{r}(\mathbf{r}(\alpha_9, \mathsf{InfList}(\mathsf{Nat}))) \\ \alpha_7 = & \mathsf{Nat} \\ \alpha_8 = & \alpha_9 \\ \alpha_{10} = & \mathsf{Nat} \\ \alpha_{11} = & \mathsf{Nat} \to \alpha_9 \\ \alpha_{12} = & \mathsf{Nat} \\ \alpha_{13} = & \mathsf{Nat} \times () \\ \alpha_{14} = & \mathsf{Nat} \\ \alpha_{15} = & () \\ X = & \alpha_9 \end{array} \right.$$

$$\alpha_2 = \mathsf{Nat}, \alpha_0 = \mathsf{InfList}(\mathsf{Nat}), \alpha_2 = \mathsf{Nat}, \forall X.\emptyset \;\Big\|\; \begin{array}{rl} \alpha_1 = & \mathbf{o}(\mathsf{Nat}) \to \mathsf{InfList}(\mathsf{Nat}) \\ \alpha_3 = & \mathsf{Nat} \\ \alpha_4 = & \bar{\mathbf{b}}\mathbf{r}(\mathbf{r}(\alpha_9, \mathsf{InfList}(\mathsf{Nat}))) \\ \alpha_5 = & \mathsf{InfList}(\mathsf{Nat}) \\ \alpha_7 = & \mathsf{Nat} \\ \alpha_8 = & \alpha_9 \\ \alpha_{10} = & \mathsf{Nat} \\ \alpha_{11} = & \mathsf{Nat} \to \alpha_9 \\ \alpha_{12} = & \mathsf{Nat} \\ \alpha_{13} = & \mathsf{Nat} \times () \\ \alpha_{14} = & \mathsf{Nat} \\ \alpha_{15} = & () \\ X = & \alpha_9 \end{array}$$

$$\alpha_2 = \mathsf{Nat}, \alpha_0 = \mathsf{InfList}(\mathsf{Nat}), \alpha_2 = \mathsf{Nat} \;\Big\|\; \begin{array}{rl} \alpha_1 = & \mathbf{o}(\mathsf{Nat}) \to \mathsf{InfList}(\mathsf{Nat}) \\ \alpha_3 = & \mathsf{Nat} \\ \alpha_4 = & \bar{\mathbf{b}}\mathbf{r}(\mathbf{r}(\alpha_9, \mathsf{InfList}(\mathsf{Nat}))) \\ \alpha_5 = & \mathsf{InfList}(\mathsf{Nat}) \\ \alpha_7 = & \mathsf{Nat} \\ \alpha_8 = & \alpha_9 \\ \alpha_{10} = & \mathsf{Nat} \\ \alpha_{11} = & \mathsf{Nat} \to \alpha_9 \\ \alpha_{12} = & \mathsf{Nat} \\ \alpha_{13} = & \mathsf{Nat} \times () \\ \alpha_{14} = & \mathsf{Nat} \\ \alpha_{15} = & () \end{array}$$

$$\alpha_0 = \mathsf{InfList}(\mathsf{Nat}) \;\Big\|\; \begin{array}{rl} \alpha_1 = & \mathbf{o}(\mathsf{Nat}) \to \mathsf{InfList}(\mathsf{Nat}) \\ \alpha_2 = & \mathsf{Nat} \\ \alpha_3 = & \mathsf{Nat} \\ \alpha_4 = & \bar{\mathbf{b}}\mathbf{r}(\mathbf{r}(\alpha_9, \mathsf{InfList}(\mathsf{Nat}))) \\ \alpha_5 = & \mathsf{InfList}(\mathsf{Nat}) \\ \alpha_7 = & \mathsf{Nat} \\ \alpha_8 = & \alpha_9 \\ \alpha_{10} = & \mathsf{Nat} \\ \alpha_{11} = & \mathsf{Nat} \to \alpha_9 \\ \alpha_{12} = & \mathsf{Nat} \\ \alpha_{13} = & \mathsf{Nat} \times () \\ \alpha_{14} = & \mathsf{Nat} \\ \alpha_{15} = & () \end{array}$$

As termination successfully finished, we can see that the result of the **unfold** term is inferred to be of type $\mathsf{InfList}(\mathsf{Nat})$.

3.6.5 Correctness of type inference

In this section we show the correctness of the type inference system in relation to the typing rules introduced in section 3.4. Due to the large number of typing rules and corresponding type inference rules, a full proper proof linking all typing rules with all type inference rules would be onerous to list here, especially since type inference is not the major focus of the thesis. Instead, we give a sketch of the structure of the proof, and follow by giving some general properties about the relationship between the two.

Proposition 1. *For any Compositional Pola term, t, and any valid Compositional Pola environments, Ξ, Γ, Δ, if $\Xi \| \Gamma \mid \Delta \vdash t : \alpha$ generates the type equation Φ and $\Phi_\alpha \rightsquigarrow \alpha'$, then there is a valid type checking derivation of $\Xi \| \Gamma \mid \Delta \vdash t : \alpha'$.*

Proof. For axiomatic rules of inference, those with no premise, there is also no quantifier introduced for these rules, so the correspondence with type checking rules is clear from inspection. For example, the type inference rule $\Xi \| \Gamma \mid \Delta \vdash () : \alpha$ generates the equation $\Phi = (\alpha = ())$. Unification yields $\Phi \rightsquigarrow (\alpha = ())$ and hence $\alpha' = ()$. The type checking rule $\Xi \| \Gamma \mid \Delta \vdash () : ()$ is a complete derivation. Other axiomatic rules follow similarly.

For all rules of inference with premises and existential quantifiers only, each premise is assigned an existentially quantified type and the correspondence with type checking follows via induction. For instance, the type inference rule $\Xi \| \Gamma \mid \Delta \vdash t \times u : \alpha$ yields the type equation $\Phi = (\exists\beta.\exists\gamma.\alpha = \beta \times \gamma)$. Following unification, we will have $\Phi \rightsquigarrow (\alpha = \beta' \times \gamma'), \Psi$ where, by induction, $\beta' \times \gamma'$ are valid in a type checking derivation. Other rules with only existential quantifiers follow similarly.

In the case of the **fold** type inference rule, the generated universally quantified equation of $\forall X.(\mathbf{r}(X, \beta) \to \beta \to \gamma) \to \mathbf{o}(\beta) \to \gamma$ matches the type given for the type checking rule, of $(\mathbf{r}(\beta, \alpha) \to \alpha \to \gamma) \to \mathbf{o}(\alpha) \to \gamma$ with the constraint that β must not match any other fold variable. Due to the resolution of universal quantifiers during the unification process, a successful unification means that the variable X in the type inference rule cannot be matched with *any* other type and thus it cannot match with any other **fold** type variable. Therefore, there is a valid derivation under the type checking rules.

In the case of the **unfold** type inference rules, we combine the rules of type-checking for **unfold** per se and for the coinductive recursive lambda construct.

- In the case of a nullary **unfold** construct, $t \equiv (\mathbf{unfold}\ (\lambda^{\bar{f}} g.u))$, the type of **unfold** per se is $(\mathbf{r}(\alpha, \beta) \to \bar{\mathbf{br}}(\mathbf{r}(\alpha, \beta))) \to \beta$. The term $(\lambda^{\bar{f}} g.u)$ has a type of $\mathbf{r}(\alpha, \beta) \to \gamma$ via the coinductive recursive lambda rule. As the term is well-typed, γ is thus necessarily of type $\bar{\mathbf{br}}(\mathbf{r}(\alpha, \beta))$ and hence the type of t is β.

  The type inference rule for nullary **unfold** assigns g the same type and assures that the term u has the appropriate type of $\bar{\mathbf{br}}(\mathbf{r}(\alpha, \beta))$ where α cannot be matched with any other type. The ultimate type of the term t is thus β.

- For an **unfold** of arity n ($n > 0$), we can assume via induction that the type inference process produces correct typing for an **unfold** of arity $n-1$. The **unfold** rule for arity $n-1$ generates type equations of the following form:

  $$\exists\beta_1, \ldots, \beta_{n-1}, \gamma, \delta\eta.\forall X.\delta = \beta_0 \to \cdots \to \beta_{n-1} \to X, \gamma = \bar{\mathbf{br}}(\mathbf{r}(X, \eta)),$$
  $$\alpha = \mathbf{o}(\beta_1) \to \cdots \to \mathbf{o}(\beta_{n-1}) \to \eta$$

The corresponding type-checking rule for an **unfold** of arity $n-1$ is as follows:

$$\begin{aligned}\mathbf{unfold} : (\mathbf{r}(\alpha, \gamma_1 \to \cdots \to \gamma_{n-1} \to \beta) \to \gamma_1 \to \cdots \to \gamma_{n-1} \to \\ \bar{\mathbf{b}}\mathbf{r}(\mathbf{r}(\alpha, \gamma_1 \to \cdots \to \gamma_{n-1} \to \beta))) \to \mathbf{o}(\gamma_1) \to \cdots \to \mathbf{o}(\gamma_{n-1}) \to \beta\end{aligned}$$

We extend the generated type equations and type-checking formulas to include the extra **unfold** parameter. The generated type equations for arity n then are as follows:

$$\begin{aligned}&\exists \beta_1, \ldots, \beta_n, \gamma, \delta\eta. \forall X. \delta = \beta_0 \to \cdots \to \beta_n \to X, \gamma = \bar{\mathbf{b}}\mathbf{r}(\mathbf{r}(X, \eta)), \\ &\alpha = \mathbf{o}(\beta_1) \to \cdots \to \mathbf{o}(\beta_n) \to \eta\end{aligned}$$

The type-checking rule for arity n is as follows:

$$\begin{aligned}\mathbf{unfold} : (\mathbf{r}(\alpha, \gamma_1 \to \cdots \to \gamma_n \to \beta) \to \gamma_1 \to \cdots \to \gamma_n \to \\ \bar{\mathbf{b}}\mathbf{r}(\mathbf{r}(\alpha, \gamma_1 \to \cdots \to \gamma_n \to \beta))) \to \mathbf{o}(\gamma_1) \to \cdots \to \mathbf{o}(\gamma_n) \to \beta\end{aligned}$$

With an extra opponent argument applied to the **unfold**, the generated type equations will then match the type-checking formulas.

As this concludes all cases, the type inference equations soundly determine a correct type for a Compositional Pola term according to section 3.4. □

Chapter 4

Pola implementation

FIXME: write this.

Chapter 5

Bounds inference

The primary motivation and contribution of this thesis is the automatic, efficient inference of time bounds. In this section we describe the mechanism by which we may efficiently infer upper bounds on running time for any well-typed Pola program.

Through this section, the reader may consider that a somewhat simpler bounds inference system could be constructed if we were satisfied with very loose bounds. However, here we provide a bounds inference system that provides bounds tight enough to be of practical use.

5.1 Sizes

Before considering time bounds, we necessarily need to define the size of a term. Space bounds of a function will be polynomials parameterized over the sizes of the parameters of that function and it is important to define the sizes appropriately, especially in the interest of obtaining useful time bounds. Consider a list of natural numbers, $[4, 756, 0]$. How we represent the size of this list will impact the bounds we can infer for differing type of operations. If we care only to compute the length of the list, the large number of 756 carries no importance. However, the magnitude of the elements of the list is more important than the length of the list if we are computing the summation of the elements of the list. With a goal of providing bounds of practical use, representing the size of such a list as a single number would not suffice. Rather than rely on a scalar representation, here we present here a more structured representation of sizes suitable for any use in Pola programs.

There are eight general forms to describe the size of a term, denoted Φ. The notation $\mathbb{N}[V]$ is used to denote polynomials over variables in V with coefficients which are natural numbers. The forms of Φ are as follows:

$$f = \lambda^p x.\mathsf{Cons}\ ((\mathsf{Zero}\ ()) \times x)$$

Figure 5.1: A short compositional Pola function which prefixes its argument, a list of natural numbers, with the number 0.

Zero	$\Phi :=$	0
Constructor	$\Phi :=$	$\langle \mathsf{C}_i : \mathbb{N}[\mathrm{V}], \Phi \mid \Phi \rangle$
Destructor	$\Phi :=$	Φ_D
Tuple	$\Phi :=$	$\Phi \times \Phi$
Polynomial	$\Phi :=$	$\mathbb{N}[\mathrm{V}]$
Variable	$\Phi :=$	Φ_v
Lambda	$\Phi :=$	$\lambda v.\Phi$
Destructor application	$\Phi :=$	$\Phi.\mathsf{D}_i$

The destructor sizes, denoted Φ_D, are represented as follows:

Recursive destructor	$\Phi_D :=$	$\lambda^{\mathsf{D}_i} \lambda y.\Phi_D$
Non-recursive destructor	$\Phi_D :=$	$\langle \mathsf{D}_i : \Phi \mid \Phi_D \rangle$

Finally, variable sizes, Φ_v, are represented as follows:

Variable name	$\Phi_v :=$	v
Constructor count	$\Phi_v :=$	$\Phi_v.\mathsf{C}_i$
Tuple projection	$\Phi_v :=$	$\Phi_v.\mathbb{N}$

Consider the example given in figure 5.1. Given a list x, it will construct the list $\mathsf{Cons}(0, x)$. The size of the term returned by the function is $\lambda x.\langle \mathsf{Nil} : 1, 0 \mid \langle \mathsf{Cons} : 1 + x.\mathsf{Cons}, (\langle \mathsf{Zero} : 1, 0 \mid \langle \mathsf{Succ} : x.\mathsf{Cons}.0.\mathsf{Succ}, 0 \rangle \rangle \times 0) \rangle \rangle$, indicating a list with exactly one Nil constructor and a number of Cons constructors one more than that of its input parameter x. The greatest element of the list returned will be no greater than the greatest element in the input list. The inferred bound in this case, using the system of inference described in section 5.1.2, is the same as the actual bound, though it should be noted that the bounds we will eventually infer will, in general, not be exactly this precise. In the interest of brevity and clarity, we will rewrite bounds to elide "0" sizes where they are not interesting and to compress chained $\langle \cdot \rangle$ notations. The bound of the function, f, then, would be written as $\lambda x.\langle \mathsf{Nil} : 1 \mid \mathsf{Cons} : 1 + x.\mathsf{Cons}, \langle \mathsf{Zero} : 1 \mid \mathsf{Succ} : x.\mathsf{Cons}.0.\mathsf{Succ} \rangle \rangle$.

The intuition behind this size is that it keeps a count of the produced list—the produced list will have 1 Nil constructor and 1 more Cons than the list x did—and the size of the each natural number in the list is bounded from above by the maximum size of the natural numbers of the list x. Ultimately, the sizes are upper bounds on counts of the constructors present in the data.

5.1.1 Operations on sizes

Before considering the mechanism of size inference, we need to consider algebraic operations on sizes. The process of inference will require us to perform many arithmetic operations on sizes. We use the notation $u[t/x]$ to stand for the term u with the term t substituted for the term x.

Addition

The only possible sizes for a term with arrow type are polynomial size, lambda size or destructor application size. If both operands are of lambda size, $\lambda x.t$ and $\lambda y.u$, then we define their summation to be $|\lambda x.t| + |\lambda y.u| = \lambda x.|t| + |u[x/y]|$; if one operand is of lambda size, then their summation is $|\lambda x.t| + u = \lambda x.(|t| + |u|)$; if neither is of lambda size, then they must either be zero or polynomial size, and addition is then defined to be the usual polynomial addition.

If a term has variable type, it cannot have any constructor, destructor or tuple information associated with it or else type inference would have unified it to a non-variable type. It must be either the zero size, a destruction size, or a polynomial size over variables. We take the size of 0 to be the polynomial 0. Addition is then defined to be usual polynomial addition.

If a term has tuple type, then it cannot have any constructor or destructor information or else it would not unify with tuple type. A term of tuple type has either zero size, polynomial size, a destructor application size or a tuple size. For any given size for a term of tuple type, α, we define $(\alpha)_i$ for $i \in \mathbb{N}$, as follows:

$$\begin{aligned}(0)_i &= 0.i \\ (v)_i &= v.i \\ (\Phi.\mathsf{D}_j)_i &= \Phi.\mathsf{D}_j.i \\ (\alpha \times \beta)_0 &= \alpha \\ (\alpha \times \beta)_{n+1} &= \beta_n\end{aligned}$$

In other words, the $(\alpha)_i$ operation is a projection of the ith component of a tuple size. Where term a and b both have tuple type of arity n, we define $|a| + |b| = (|a|_1 + |b|_1) \times ((|a|_2 + |b|_2) + \cdots + (|a|_n + |b|_n)))$.

If a term has inductive type, then it has zero size, destructor application size, polynomial size or constructor size. For any given size for a term of inductive type, α, we define $(\alpha)_{\mathsf{C}_i}$ for any constructor C_i of the appropriate type, as follows:

$$\begin{aligned}(0)_{\mathsf{C}_i} &= 0.\mathsf{C}_i \\ (v)_{\mathsf{C}_i} &= v.\mathsf{C}_i \\ (\Phi.\mathsf{D}_j)_{\mathsf{C}_i} &= \Phi.\mathsf{D}_j.\mathsf{C}_i \\ (\langle \mathsf{C}_i : x, x' \mid \Phi \rangle)_{\mathsf{C}_i} &= x_i \\ (\langle \mathsf{C}_j : x, x' \mid \Phi \rangle)_{\mathsf{C}_i} &= (\Phi)_{\mathsf{C}_i}\end{aligned}$$

The $(\alpha)_{\mathsf{C}_i}$ thus gives us the number of constructors of a particular value. For determining the maximum size of data associated with a constructor, we define $(\alpha)'_{\mathsf{C}_i}$ as follows:

$$\begin{aligned}(0)'_{\mathsf{C}_i} &= 0.\mathsf{C}'_i \\ (v)'_{\mathsf{C}_i} &= v.\mathsf{C}'_i \\ (\Phi.\mathsf{D}_j)'_{\mathsf{C}_i} &= \Phi.\mathsf{D}_j.\mathsf{C}'_i \\ (\langle \mathsf{C}_i : x, x' \mid \Phi \rangle)'_{\mathsf{C}_i} &= x'_i \\ (\langle \mathsf{C}_j : x, x' \mid \Phi \rangle)'_{\mathsf{C}_i} &= (\Phi)'_{\mathsf{C}_i}\end{aligned}$$

The only difference between $(\alpha)_{\mathsf{C}_i}$ and $(\alpha)'_{\mathsf{C}_i}$ for sizes of values with inductive type is whether to evaluate to x_i or x'_i. For instance, the list $t = [3, 5]$ has size $|t| = \langle \mathsf{Nil} : 1 \mid \mathsf{Cons} : 2, \langle \mathsf{Zero} : 1 \mid \mathsf{Succ} : 5 \rangle \rangle$. $(|t|)_{\mathsf{Cons}} = 2$ and $(|t|)'_{\mathsf{Cons}} = \langle \mathsf{Zero} : 1 \mid \mathsf{Succ} : 5 \rangle$.

For any two terms, a and b, of the same inductive type with n constructors $\mathsf{C}_1, \ldots, \mathsf{C}_n$, we define addition as follows:

$$|a| + |b| = \langle \mathsf{C}_1 : (|a|)_{\mathsf{C}_1} + (|b|)_{\mathsf{C}_1}, \max((|a|)'_{\mathsf{C}_1}, (|b|)'_{\mathsf{C}_1}) \mid \cdots \mid \mathsf{C}_n : (|a|)_{\mathsf{C}_n} + (|b|)_{\mathsf{C}_n}, \max((|a|)'_{\mathsf{C}_n}, (|b|)'_{\mathsf{C}_n}) \rangle$$

Subtraction

The operation of subtraction is required in the case of inferring recursive coinductive sizes. This carries out similarly to addition, with coefficients of the subtrahend being negated in the obvious way. In practice we will use a monus operation, wherein the results are constrained to be non-negative, denoted $\overset{\circ}{-}$. Consider the following example to demonstrate:

$$\langle \mathsf{A} : x^2 + 2x + 4 \mid \mathsf{B} : x + 2 \rangle \overset{\circ}{-} \langle \mathsf{A} : x^2 + x + 1 \mid \mathsf{B} : x^2 + 3 \rangle = \langle \mathsf{A} : x + 3 \mid \mathsf{B} : x \rangle$$

In this case we can see an element-wise subtraction of terms, such that any negative term in the difference is replaced with 0.

Multiplication

All sizes may be multiplied by a polynomial size, as by repeated addition by terms.

Maximum

Due to the novel structure of sizes, we introduce a novel definition of a maximum function over size values. Unlike typical maximum functions, it is not always the case that $\max(a, b) \in \{a, b\}$. It is always the case that $\max(a, b) \leq a + b$, however. Consider $\max(a^2 + b, 2b)$, which we define to be $a^2 + 2b$ in the absence of any known values for variables a and b. We see that $a^2 + b < \max(a^2 + b, 2b)$ and $2b < \max(a^2 + b, 2b)$ and $\max(a^2 + b, 2b) < a^2 + b + 2b$.

Polynomial maximum between two polynomials $p, q \in \mathbb{N}[\mathbb{V}]$, where each polynomial is a summation of terms, namely $p = \sum_{i=1}^{n_p} a_i \prod_{j=1}^{n_{p_i}} v_j^{p_{i,j}}$ and $q = \sum_{i=1}^{n_q} b_i \prod_{j=1}^{n_{q_i}} v_j^{q_{i,j}}$ where each $p_{i,j}, q_{i,j}, a_i, b_i \in \mathbb{N}$, is defined as follows:

$$\max(p, q) = \sum_{i, \forall j. p_{i,j} = q_{i,j}} \max(a_i, b_i) + \sum_{i, \exists j. p_{i,j} \neq q_{i,j}} (a_i \prod_{j=1}^{n_{p_i}} v_j^{p_{i,j}} + b_i \prod_{j=1}^{n_{q_i}} v_j^{q_{i,j}})$$

In other words, we take the maximum of the coefficients for like terms; for unlike terms, we simple add them. As an example, $\max(3a^2+3ab+6, 2a^2+4a^2b+1) = 3a^2+4a^2b+3ab+6$.

If the two terms have arrow type, they must be of polynomial size, lambda size or destruction application size. If they both have lambda sizes, $\lambda x.t$ and $\lambda y.u$, then $\max(\lambda x.t, \lambda y.u) = \lambda x. \max(t, u[x/y])$. If one term has lambda size, $\lambda x.t$ and the other term has size u, then $\max(\lambda x.t, u) = \lambda x. \max(t, u)$. If neither has lambda size, then they have zero, destruction application or polynomial size and maximum is defined as polynomial maximum.

If the two terms have variable type, then they have zero, destructor application or polynomial size and maximum is defined as polynomial maximum.

If the two terms have tuple type, then they have zero, destructor application, polynomial or tuple size. Similarly to defining addition, we define $\max(a, b) = \max(|a|_1, |b|_1) \times \max(|a|_2, |b|_2) \times \cdots \times \max(|a|_n, |b|_n)$.

If the two terms have inductive type, then they have zero, destructor application, polynomial or inductive size. Similarly to defining addition, we define maximum for inductive sizes as follows:

$$\begin{aligned}\max(a, b) = \langle \mathsf{C}_1 : \max((|a|)_{\mathsf{C}_1}, (|b|)_{\mathsf{C}_1}), \max((|a|)'_{\mathsf{C}_1}, (|b|)'_{\mathsf{C}_1}) \mid \cdots \mid \\ \mathsf{C}_n : \max((|a|)_{\mathsf{C}_n}, (|b|)_{\mathsf{C}_n}), \max((|a|)'_{\mathsf{C}_n}, (|b|)'_{\mathsf{C}_n}) \rangle\end{aligned}$$

Dot product

For inferring the bounds on **fold** constructs, it is necessary to perform a dot product between the size of the input and the sizes of the branches. We define dot product as follows:

$$\begin{aligned} 0 \cdot () &= 0 \\ \langle \mathsf{C}_i : x, y \mid \Phi \rangle \cdot (\Psi_1 \times \Psi_2) &= x\Psi_1 + \Phi \cdot \Psi_2 \end{aligned}$$

Application

Application of size terms is only valid between two operand sizes where the first is a lambda size. In other words, $t\ u$ is well-defined if and only if t is of the form $\lambda v.\Phi$. The result in this case is as expected, where $(\lambda v.\Phi)\ u = \Phi[u/v]$, i.e., the size Φ with all references to the variable size v replaced by the size u.

Destructor application

Invocation of a non-recursive destructor D_i in a size t is written as $t[\mathsf{D}_i]$. This operation is defined as:

$$\begin{aligned} \langle \ldots \mid \mathsf{D}_i : \alpha \mid \ldots \rangle[\mathsf{D}_i] &\equiv \alpha \\ \alpha[\mathsf{D}_i] &\equiv \alpha.\mathsf{D}_i \end{aligned}$$

Invocation of a recursive destructor requires a modification of potentially all elements of a coinductive size. This occurs when an object is destructed via a recursive destructor. See section 5.1.2 for more detail on size bounds and recursive deconstructions. The notation $\alpha[\mathsf{D}_i/x/y]$ indicates that any reference to the recursive destructor D_i over bound variable x is replaced with size term y. Consider the following example:

$$(\lambda^{\mathsf{Tail}} \lambda x.\lambda^{\mathsf{Head}}.x + 1)[\mathsf{Tail}/z/z + 1] = \lambda^{\mathsf{Tail}} \lambda x.\lambda^{\mathsf{Head}}.x + 2$$

In this case the variable x has been replaced with the size term $x + 1$, yielding a size term of $x + 2$.

$$\text{Variable reference} \quad \frac{(x : \alpha) \in \Gamma \text{ or } (x : \alpha) \in \Delta \text{ or } (x : \alpha) \in \Xi}{\Xi \| \Gamma \mid \Delta \vdash x : \alpha}$$

$$\text{Construction} \quad \frac{\Xi \| \Gamma \mid \Delta \vdash \mathsf{C}_i : \alpha \quad \Xi \| \Gamma \mid \Delta \vdash t : \beta}{\Xi \| \Gamma \mid \Delta \vdash \mathsf{C}_i\ t : \alpha\ \beta}$$

$$\text{Opponent application} \quad \frac{\Xi \| \Gamma \mid \Delta \vdash t : \alpha_t \quad \Gamma \mid \Delta \vdash u : \alpha_u}{\Xi \| \Gamma \mid \Delta \vdash t\ (u)_{\mathbf{o}} : \alpha_t\ \alpha_u}$$

$$\text{Player application} \quad \frac{\Xi \| \Gamma \mid \Delta \vdash t : \alpha_t \quad \Gamma \mid \Delta \vdash u : \alpha_u}{\Xi \| \Gamma \mid \Delta \vdash t\ u : \alpha_t\ \alpha_u}$$

$$\text{Tuple unit} \quad \frac{}{\Xi \| \Gamma \mid \Delta \vdash () : 0}$$

$$\text{Tuple composition} \quad \frac{\Xi \| \Gamma \mid \Delta \vdash t : \alpha_t \quad \Xi \| \Gamma \mid \Delta \vdash u : \alpha_u}{\Xi \| \Gamma \mid \Delta \vdash t \times u : \alpha_t \times \alpha_u}$$

$$\text{Opponent lambda} \quad \frac{\Xi \| (x : x), \Gamma \mid \Delta \vdash t : \alpha_t}{\Xi \| \Gamma \mid \Delta \vdash \lambda^o x.t : \lambda x.\alpha_t}$$

$$\text{Player lambda} \quad \frac{\Xi \| \Gamma \mid (x : x), \Delta \vdash t : \alpha_t}{\Xi \| \Gamma \mid \Delta \vdash \lambda^p x.t : \lambda x.\alpha_t}$$

Figure 5.2: Basic rules of size inference.

5.1.2 Inferring size bounds

Figure 5.2 gives rules of inference for defining sizes of simple Compositional Pola terms. Each sequent is of the form $\Xi \| \Gamma \mid \Delta \vdash t : \alpha$ where Ξ, Γ, Δ are mappings of symbol names to sizes, t is a term and α is the size of term t. In all of these cases, the resultant size is a direct reflection of the term: in effect the size is an encoding of the structure of the term. In the interest of tight size bounds, we avoid any sort of contraction of term structure wherever possible.

In the case of control structures such as **peek** and **case** constructs, we cannot effectively encode the structure of term comprehensively in the size because the value of the subject is unknown. The rules of size inference for these terms is given in figure 5.3. As we can see in the branch composition rule, we use the max operation to get a bound on these terms.

The rules for constructor matching and variable matching require special explanation. They use the two special variables C and N, which we consider to be unique from any other variable. The variable C is used to indicate the subject of a **peek** or **case** as it is currently being considered. The variable N is used to indicate the current index of the variable to be considered next. For example, consider the term **peek** x $(\lambda^{\mathsf{C}}\mathsf{Nil}.\triangledown.t + \lambda^{\mathsf{C}}\mathsf{Cons}.\Diamond.\lambda^{p}y.\Diamond.\lambda^{p}z.\triangledown.u)$. When performing size inference of the branch $\lambda^{\mathsf{C}}\mathsf{Nil}.\triangledown.t$, we will consider the size of C to be $C.\mathsf{Nil}$ and N will never be anything other than 0. When performing size inference on the branch $\lambda^{\mathsf{C}}\mathsf{Cons}.\Diamond.\lambda^{p}y.\Diamond.\lambda^{p}z.\triangledown.u$, we will consider the size of C to be $C.\mathsf{Cons}$. Initially the value of N will be 0, yielding a size for y of $C.\mathsf{Cons}.0$. N will take on the value of 1, though there are no further non-recursive variables that are introduced. Since the variable z is recursive—i.e., of the same type as the term being peeked in itself—it takes on the size of $C.\mathsf{Cons}$.

Folds

Inferring size bounds for **fold** constructs is a two-stage process. We first infer the size of the branches with the recursive function, f, producing zero-sized values: this allows us to determine the fixed size of the value produced by the **fold**. We then infer the size of the branches again, but with the recursive function, f, producing values of the size indicated in the first step, and with the bound variables having zero size: this allows us to determine the increase in size of the value produced by the **fold** each time a recursion is performed. From these two steps we get two sizes: the fixed size and the increase in size per recursion. A bound on the ultimate size of the value produced by the **fold** is then an affine function over the size of the input, with these two values as coefficients.

Figure 5.4 gives the rule of inference for inferring sizes from **fold** constructs. Within the sequents, we use the notation $\Xi \| \Gamma \mid \Delta \vdash \overline{t} : \alpha$ to indicate that we are inferring in stage 1; we use the notation $\Xi \| \Gamma \mid \Delta \vdash \underline{t} : \alpha$ to indicate that we are inferring in stage 2. The α that is tagged with the overline indicates the name of the recursive function (or "0" if the name is not known yet); the α that is tagged with the underline indicates the size the returned value of the recursive function (or "0" if it is no longer needed).

An example of inductive recursion size bound inference is given in figure 5.11 gives an example derivation of size bounds on the function *add* given in figure 3.3. The resultant

$$\text{Peek}\quad \frac{\Xi \| \Gamma \mid \Delta \vdash b : \alpha_b \quad \Xi \| \Gamma \mid (C : \alpha_b)\Delta \vdash t : \alpha_t}{\Xi \| \Gamma \mid \Delta \vdash \mathbf{peek}\ t\ b : \alpha_b\ \alpha_t}$$

$$\text{Case}\quad \frac{\Xi \| \Gamma \mid \Delta \vdash b : \alpha_b \quad \Xi \| \Gamma \mid (C : \alpha_b)\Delta \vdash t : \alpha_t}{\Xi \| \Gamma \mid \Delta \vdash \mathbf{case}\ t\ b : \alpha_b\ \alpha_t}$$

$$\text{Branch composition}\quad \frac{\Xi \| \Gamma \mid \Delta \vdash b_1 : \alpha_1 \quad \Xi \| \Gamma \mid \Delta \vdash b_2 : \alpha_2}{\Xi \| \Gamma \mid \Delta \vdash b_1 + b_2 : \max(\alpha_1, \alpha_2)}$$

$$\text{Constructor match}\quad \frac{\Xi \| \Gamma \mid (C : C.\mathsf{C}_i), (N : -1), \Delta \vdash t : \alpha_t}{\Xi \| \Gamma \mid (C : C), \Delta \vdash \lambda^{\mathsf{C}}\mathsf{C}_i.t : \lambda C.\alpha_t}$$

$$\text{Opponent variable match}\quad \frac{\Xi \| \Gamma \mid (C : \alpha_C), (x : \alpha_C), \Delta \vdash t : \alpha_t}{\Xi \| \Gamma \mid (C : \alpha_C), \Delta \vdash \Diamond.\lambda^o x.t : \alpha_t}\ x \text{ is recursive}$$

$$\text{Opponent variable match}\quad \frac{\Xi \| (x : \alpha_C.n), \Gamma \mid (C : \alpha_C), (N : n+1), \Delta \vdash t : \alpha_t}{\Xi \| \Gamma \mid (C : \alpha_C), (N : n), \Delta \vdash \Diamond.\lambda^o x.t : \alpha_t}\ \text{otherwise}$$

$$\text{Player variable match}\quad \frac{\Xi \| \Gamma \mid (C : \alpha_C), (x : \alpha_C)\Delta \vdash t : \alpha_t}{\Xi \| \Gamma \mid (C : \alpha_C), \Delta \vdash \Diamond.\lambda^p x.t : \alpha_t}\ x \text{ is recursive}$$

$$\text{Player variable match}\quad \frac{\Xi \| \Gamma \mid (C : \alpha_C), (N : n+1), (x : \alpha_C.n), \Delta \vdash t : \alpha_t}{\Xi \| \Gamma \mid (C : \alpha_C), (N : n), \Delta \vdash \Diamond.\lambda^p x.t : \alpha_t}\ \text{otherwise}$$

$$\text{Fold variable match}\quad \frac{\Xi \| (x : \alpha_C), \Gamma \mid (y : \alpha_C), \Delta \vdash t : \alpha_t}{\Xi \| \Gamma \mid (C : \alpha_C), \Delta \vdash \lambda^{op} xy.t : \alpha_t}$$

$$\text{Unit match}\quad \frac{\Xi \| \Gamma \mid \Delta \vdash t : \alpha_t}{\Xi \| \Gamma \mid (C : c), (N : n), \Delta \vdash \nabla.t : \alpha_t}$$

Figure 5.3: Rules of size inference for case constructs.

$$\text{Fold}\quad \frac{\Xi\|\Gamma \mid \Delta \vdash \overline{t}^{0} : \beta \quad \Xi\|\Gamma \mid \Delta \vdash \underline{t}_{\beta} : \gamma \quad \Xi\|\Gamma \mid \Delta \vdash u : \delta}{\Xi\|\Gamma \mid \Delta \vdash \mathbf{fold}\ t\ u : \delta \cdot \gamma + \beta}$$

$$\text{Recursive lambda (pass 1)}\quad \frac{\Xi, (f : 0)\|\Gamma \mid \Delta \vdash \overline{t}^{f} : \alpha}{\Xi\|\Gamma \mid \Delta \vdash \overline{\lambda^{f} f.t}^{0} : \alpha}$$

$$\text{Fold parameter (pass 1)}\quad \frac{\Xi, (f : x \to \beta)\|\Gamma \mid \Delta, (x : x) \vdash \overline{t}^{f} : \alpha}{\Xi, (f : \beta)\|\Gamma \mid \Delta \vdash \overline{\lambda^{p} x.t}^{f} : \alpha}$$

$$\text{Fcase (pass 1)}\quad \frac{\Xi\|\Gamma \mid \Delta \vdash x : \beta_x \quad \Xi\|\Gamma \mid (C : \beta_x), \Delta \vdash b : \alpha}{\Xi\|\Gamma \mid \Delta \vdash \overline{\mathbf{fcase}\ x\ b}^{f} : \alpha}$$

$$\text{Recursive lambda (pass 2)}\quad \frac{\Xi, (f : \beta)\|\Gamma \mid \Delta \vdash \underline{t}_{0} : \alpha}{\Xi\|\Gamma \mid \Delta \vdash \underline{\lambda^{f} f.t}_{\beta} : \alpha}$$

$$\text{Fold parameter (pass 2)}\quad \frac{\Xi, (f : x \to \gamma)\|\Gamma \mid \Delta, (x : 0) \vdash \underline{t}_{0} : \alpha}{\Xi, (f : \gamma)\|\Gamma \mid \Delta \vdash \underline{\lambda^{p} x.t}_{0} : \alpha}$$

$$\text{Fcase (pass 2)}\quad \frac{\Xi\|\Gamma \mid \Delta \vdash x : \beta_x \quad \Xi\|\Gamma \mid (C : \beta_x), \Delta \vdash b : \alpha}{\Xi\|\Gamma \mid \Delta \vdash \underline{\mathbf{fcase}\ x\ b}_{0} : \alpha}$$

Figure 5.4: Rules of size inference for **fold** constructs.

size bound inferred is $\langle \mathsf{Zero} : y.\mathsf{Zero} \mid \mathsf{Succ} : x.\mathsf{Succ} + y.\mathsf{Succ} \rangle$, indicating that the value produced by adding the two natural numbers x and y is a number number containing, at most, the same number of Zero constructors as y did and containing, at most, a number of Succ constructors equal to the sum of the two operands. Since every natural number has exactly one Zero constructor, the resultant value has exactly one Zero constructor.

As a concrete example, consider that we wished to add the numbers $x = \bar{4}$ and $y = \bar{9}$. x has size $\langle \mathsf{Zero} : 1 \mid \mathsf{Succ} : 4 \rangle$ and y has size $\langle \mathsf{Zero} : 1 \mid \mathsf{Succ} : 9 \rangle$. The inferred bound tells that the *add* function applies to these two operands will yield a value of at most $\langle \mathsf{Zero} : 1 \mid \mathsf{Succ} : 13 \rangle$, i.e., the number $\overline{13}$.

It is clear to see that in all cases of the *add* function, we yield a natural number with a number of Succ constructors equal to the sum of its two operands. Consequently, in this case, the inference process has inferred a size metric which is exactly right for all inputs. In general we cannot assume the bound inferred to be exactly accurate, but it will be a valid upper bound.

$$\dfrac{\dfrac{}{(f : \lambda x.\lambda y.y) \| (x : 0) \mid (y : 0), (z : 0) \vdash y : \langle \mathsf{Zero} : 0 \mid \mathsf{Succ} : 0 \rangle}}{(f : \lambda x.\lambda y.y) \| (x : 0) \mid (y : 0), (z : 0) \vdash \mathsf{Succ}\ y : \langle \mathsf{Zero} : 0 \mid \mathsf{Succ} : 1 \rangle}$$

Figure 5.5: A continuation of the derivation given in figure 5.6.

$$\dfrac{\dfrac{\dfrac{}{(f : \lambda x.\lambda y.y) \| \cdots \mid \cdots \vdash f : \lambda x.\lambda y.y} \quad \dfrac{}{\cdots \| \cdots \mid (y : 0), (z : 0) \vdash z : 0}}{(f : \lambda x.\lambda y.y) \| (x : 0) \mid (y : 0), (z : 0) \vdash f\ z : \lambda y.y} \quad \text{Continued in figure 5.5}}{(f : \lambda x.\lambda y.y) \| (x : 0) \mid (y : 0), (z : 0) \vdash f\ z\ (\mathsf{Succ}\ y) : \langle \mathsf{Zero} : 0 \mid \mathsf{Succ} : 1 \rangle}$$

Figure 5.6: A continuation of the derivation given in figure 5.7.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(f : \lambda x.\lambda y.y) \| (x : 0) \mid (y : 0) \vdash y : 0}}{(f : \cdots) \| (x : 0) \mid \begin{matrix}(C : 0), (N : -1),\\ (y : 0)\end{matrix} \vdash \triangledown.y : 0}}{(f : \cdots) \| (x : 0) \mid (C : 0), (y : 0) \vdash \lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.y : 0} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(f : \cdots) \| (x : 0) \mid \begin{matrix}(C : 0), (N : 0),\\ (y : 0), (z : 0)\end{matrix} \vdash \triangledown.f\ z\ (\mathsf{Succ}\ y) : \langle \mathsf{Zero} : 0 \mid \mathsf{Succ} : 1 \rangle}}{(f : \cdots) \| (x : 0) \mid \begin{matrix}(C : 0), (N : 0),\\ (y : 0)\end{matrix} \vdash \lambda^{op} - z.\triangledown.f\ z\ (\mathsf{Succ}\ y) : \langle \mathsf{Zero} : 0 \mid \mathsf{Succ} : 1 \rangle}}{(f : \cdots) \| (x : 0) \mid \begin{matrix}(C : 0), (N : -1),\\ (y : 0)\end{matrix} \vdash \Diamond.\lambda^{op} - z.\triangledown.f\ z\ (\mathsf{Succ}\ y) : \langle \mathsf{Zero} : 0 \mid \mathsf{Succ} : 1 \rangle}}{\cdots \| (x : 0) \mid (C : 0), (y : 0) \vdash \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} - z.\triangledown.f\ z\ (\mathsf{Succ}\ y) : \langle \mathsf{Zero} : 0 \mid \mathsf{Succ} : 1 \rangle} \quad \text{Continued in figure 5.6}}{(f : \lambda x.\lambda y.y) \| (x : 0) \mid (C : 0), (y : 0) \vdash \begin{matrix}\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.y\\ + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} - z.\triangledown.f\ z\ (\mathsf{Succ}\ y)\end{matrix} : \langle \mathsf{Zero} : 0 \mid \mathsf{Succ} : \langle \mathsf{Zero} : 0 \mid \mathsf{Succ} : 1 \rangle \rangle}$$

Figure 5.7: A continuation of the derivation given in figure 5.8.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{(f : \lambda x.\lambda y.y) \| (x : 0) \mid (y : 0) \vdash x : 0} \quad \text{Continued in figure 5.7}}{(f : \lambda x.\lambda y.y) \| (x : 0) \mid (y : 0) \vdash \underline{\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\nabla.y + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} - z.\nabla.f\ z\ (\mathsf{Succ}\ y))}_{0} : \langle \mathsf{Zero} : 0 \mid \mathsf{Succ} : \langle \mathsf{Z} : 0 \mid \mathsf{S} : 1 \rangle \rangle}}{(f : \lambda x.\lambda y.y) \| (x : 0) \mid (y : y) \vdash \underline{\lambda^{p} y.\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\nabla.y + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} - z.\nabla.f\ z\ (\mathsf{Succ}\ y))}_{0} : \lambda y.\langle \mathsf{Zero} : 0 \mid \mathsf{Succ} : \langle \mathsf{Z} : 0 \mid \mathsf{S} : 1 \rangle \rangle}}{(f : \lambda x.\lambda y.y) \| (x : x) \mid (y : y) \vdash \underline{\lambda^{p} x.\lambda^{p} y.\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\nabla.y + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} - z.\nabla.f\ z\ (\mathsf{Succ}\ y))}_{0} : \lambda x.\lambda y.\langle \mathsf{Zero} : 0 \mid \mathsf{Succ} : \langle \mathsf{Z} : 0 \mid \mathsf{S} : 1 \rangle \rangle}}{\| (x : x) \mid (y : y) \vdash \underline{\lambda^{f} f.\lambda^{p} x.\lambda^{p} y.\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\nabla.y + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} - z.\nabla.f\ z\ (\mathsf{Succ}\ y))}_{\lambda x.\lambda y.y} : \lambda x.\lambda y.\langle \mathsf{Zero} : 0 \mid \mathsf{Succ} : \langle \mathsf{Z} : 0 \mid \mathsf{S} : 1 \rangle \rangle}$$

Figure 5.8: A continuation of the derivation given in figure 5.11.

$$\dfrac{\dfrac{\dfrac{\overline{\cdots \| \ \mid (x : x), (y : y) \vdash y : y}}{\cdots \| \ \mid (C : x.\mathsf{Zero}), (N : -1), (x : x), (y : y) \vdash \nabla.y : y} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{(f : \lambda x.\lambda y.0) \| \ \mid (x : x), (y : y), (z : x.\mathsf{Succ}) \vdash f\ z\ (\mathsf{Succ}\ y) : 0}}{(f : \lambda x.\lambda y.0) \| \ \mid (C : x.\mathsf{Succ}), (N : 0), (x : x), (y : y), (z : x.\mathsf{Succ}) \vdash \nabla.f\ z\ (\mathsf{Succ}\ y) : 0}}{(f : \lambda x.\lambda y.0) \| \ \mid (C : x.\mathsf{Succ}), (N : 0), (x : x), (y : y) \vdash \lambda^{op} - z.\nabla.f\ z\ (\mathsf{Succ}\ y) : 0}}{(f : \lambda x.\lambda y.0) \| \ \mid (C : x.\mathsf{Succ}), (N : -1), (x : x), (y : y) \vdash \Diamond.\lambda^{op} - z.\nabla.f\ z\ (\mathsf{Succ}\ y) : 0}}{\cdots \| \ \mid (C : x), (x : x), (y : y) \vdash \lambda^{\mathsf{C}}\mathsf{Zero}.\nabla.y : y} \quad (f : \lambda x.\lambda y.0) \| \ \mid (C : x), (x : x), (y : y) \vdash \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} - z.\nabla.f\ z\ (\mathsf{Succ}\ y) : 0}{(f : \lambda x.\lambda y.0) \| \ \mid (C : x), (x : x), (y : y) \vdash \lambda^{\mathsf{C}}\mathsf{Zero}.\nabla.y + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} - z.\nabla.f\ z\ (\mathsf{Succ}\ y) : \max(y, 0) = y}$$

Figure 5.9: A continuation of the derivation given in figure 5.10.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{\cdots \| \ \mid (x:x),(y:y) \vdash x:x} \quad \text{Continued in figure 5.9}}{(f:\lambda x.\lambda y.0) \| \ \mid (x:x),(y:y) \vdash \overline{\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.y + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} - z.\triangledown.f\ z\ (\mathsf{Succ}\ y))}^{f} : y}}{(f:\lambda x.0) \| \ \mid (x:x),(y:y) \vdash \overline{\lambda^{p} y.\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.y + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} - z.\triangledown.f\ z\ (\mathsf{Succ}\ y))}^{f} : \lambda y.y}}{(f:0) \| (x:x) \mid (y:y) \vdash \overline{\lambda^{p} x.\lambda^{p} y.\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.y + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} - z.\triangledown.f\ z\ (\mathsf{Succ}\ y))}^{f} : \lambda x.\lambda y.y}}{\| (x:x) \mid (y:y) \vdash \overline{\lambda^{f} f.\lambda^{p} x.\lambda^{p} y.\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.y + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} - z.\triangledown.f\ z\ (\mathsf{Succ}\ y))}^{0} : \lambda x.\lambda y.y}$$

Figure 5.10: A continuation of the derivation given in figure 5.11.

$$\dfrac{\dfrac{\text{Continued in figure 5.10} \quad \text{Continued in figure 5.8} \quad \overline{\|(x:x) \mid (y:y) \vdash x:x}}{\|(x:x) \mid (y:y) \vdash \mathbf{fold}\ \begin{pmatrix} \lambda^{f} f.\lambda^{p} x.\lambda^{p} y.\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.y \\ + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} - z.\triangledown.f\ z\ (\mathsf{Succ}\ y)) \end{pmatrix}\ x : \lambda y.y + \langle \mathsf{Zero} : 0 \mid \mathsf{Succ} : x.\mathsf{Succ} \rangle} \quad \overline{\| \cdots \mid (y:y) \vdash y:y}}{\|(x:x) \mid (y:y) \vdash \mathbf{fold}\ \begin{pmatrix} \lambda^{f} f.\lambda^{p} x.\lambda^{p} y.\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.y \\ + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} - z.\triangledown.f\ z\ (\mathsf{Succ}\ y)) \end{pmatrix}\ x\ y : \langle \mathsf{Zero} : y.\mathsf{Zero} \mid \mathsf{Succ} : x.\mathsf{Succ} + y.\mathsf{Succ} \rangle}$$

Figure 5.11: Size bound inference for the *add* function given in figure 3.3.

$$\text{Destructor}\quad \frac{\Xi \| \Gamma \mid \Delta \vdash t : \alpha}{\Xi \| \Gamma \mid \Delta \vdash \lambda^{\mathsf{D}} \mathsf{D}_i.t : \langle \mathsf{D}_i : \alpha \rangle}\ \mathsf{D}_i \text{ is not recursive}$$

$$\text{Record}\quad \frac{\Xi \| \Gamma \mid \Delta \vdash t : \alpha}{\Xi \| \Gamma \mid \Delta \vdash \mathbf{record}\ t : \alpha}$$

$$\text{Co-branch composition}\quad \frac{\Xi \| \Gamma \mid \Delta \vdash t : \alpha \quad \Xi \| \Gamma \mid \Delta \vdash u : \beta}{\Xi \| \Gamma \mid \Delta \vdash t \oplus u : \langle \alpha \mid \beta \rangle}$$

$$\text{Destruction}\quad \frac{\Xi \| \Gamma \mid \Delta \vdash t : \alpha}{\Xi \| \Gamma \mid \Delta \vdash t\ \mathsf{D}_i : \alpha[\mathsf{D}_i]}$$

Figure 5.12: Rules of size inference for non-recursive coinductive size bounds.

Non-recursive coinductive sizes

For the size of an inductive term we can map the inferred size onto some real-world concept of memory usage. A value of type $\mathsf{List}(\mathsf{Bool})$ that has aggregate size 12 can be thought of as consuming more memory than a value of type $\mathsf{List}(\mathsf{Bool})$ that has size 4, for instance. The metrics for sizes used to describe the size of an inductive value is abstracted away from the underlying representation, taking a naïve idea that each constructor consumes the same amount of memory, but the asymptotic size bound inferred will directly correlate with the asymptotic memory consumption in a real-world representation.

This does not hold for coinductive types. Every value of coinductive type will have constant size and that constant size is completely dependent on the underlying representation, specifically the amount of memory it takes to represent a closure. For the scope of this document we do not wish to consider the details of the underlying representation and representing the size bounds of a coinductive value in this way would distract from our target, which is to meaningfully bound the sizes of values that arise from the interplay between inductive types and coinductive types.

A general intuition for this section is to consider that coinductive types and coinductive values have no purpose *per se*, but exist only to be later "unwound"—via a destruction operation—by an inductive term. In that light, we consider the size bounds of a coinductive term to be a *potential* size bound, i.e., the size of the term that could be if the coinductive value were destructed.

As an example, consider the coinductive product type given in figure 3.6. If one were to construct a product of that type with the Pola term $(\ \mathsf{P}_1 : \overline{3}; \mathsf{P}_2 : \overline{1}\ )$, we would consider the potential size of that term to be $\langle \mathsf{P}_1 : \langle \mathsf{Zero} : 1 \mid \mathsf{Succ} : 3 \rangle \mid \mathsf{P}_2 : \langle \mathsf{Zero} : 1 \mid \mathsf{Succ} : 1 \rangle \rangle$ to represent the size of the resultant term should that coinductive value ever be destructed with either the P_1 or P_2 destructor.

Figure 5.12 gives rules of inference for coinductive terms that do not involve an **unfold**. For the "Destruction" rule, the size $\langle \mathsf{D}_i : \alpha \mid \beta \rangle$ may not always have the D_i size first; we assume without loss of generality that the terms will be reordered appropriately.

Figure 5.13 gives an example of size bound inference for the Pola record ($\mathsf{Eval} : x.\mathsf{Succ}(x)$), which is of type $\mathsf{Fn}(\mathsf{Nat}, \mathsf{Nat})$, a type introduced in figure 3.6. We see that the term has size bound $\langle \mathsf{Eval} : \lambda x.\langle \mathsf{Zero} : x.\mathsf{Zero} \mid \mathsf{Succ} : 1 + x.\mathsf{Succ} \rangle \rangle$ to show that, if the Eval destructor were applied to the record with argument x, the result would be a term with one Succ constructor more than x.

$$\dfrac{\dfrac{\dfrac{\overline{\| \ | \ (x : x) \vdash \mathsf{Succ} : \lambda y.\langle \mathsf{Zero} : y.\mathsf{Zero} \mid \mathsf{Succ} : 1 + y.\mathsf{Succ}\rangle} \quad \overline{\| \ | \ (x : x) \vdash x : x}}{\| \ | \ (x : x) \vdash \mathsf{Succ}\ x : \langle \mathsf{Zero} : x.\mathsf{Zero} \mid \mathsf{Succ} : 1 + x.\mathsf{Succ}\rangle}}{\| \ | \vdash \lambda^{p} x.\mathsf{Succ}\ x : \lambda x.\langle \mathsf{Zero} : x.\mathsf{Zero} \mid \mathsf{Succ} : 1 + x.\mathsf{Succ}\rangle}}{\| \ | \vdash \lambda^{\mathsf{D}}\mathsf{Eval}.\lambda^{p} x.\mathsf{Succ}\ x : \langle \mathsf{Eval} : \lambda x.\langle \mathsf{Zero} : x.\mathsf{Zero} \mid \mathsf{Succ} : 1 + x.\mathsf{Succ}\rangle\rangle}$$

Figure 5.13: Size bound inference on the non-recursive coinductive Pola term ($\mathsf{Eval} : x.\mathsf{Succ}(x)$).

Recursive coinductive sizes

A recursive coinductive size—i.e., a coinductive term involving an **unfold** construct—will generally have an unbounded potential size. The potential size of the term yielded via destruction depends on the number of times it is destructed. This is most plainly seen in the *allNats* function given in figure 3.8, which produces an infinite list of all natural numbers. Destructing the value yielded by *allNats* with the Head destructor yields a term with size $\langle \mathsf{Zero} : 1 \mid \mathsf{Succ} : 0 \rangle$; destructing the value yielded by *allNats* with the Tail destructor yields another coinductive term such that destructing *that* with the Head destructor would yield a term with a larger size. In general we can say that the size of the term yielded by applying the term to the Head destructor is $\langle \mathsf{Zero} : 1 \mid \mathsf{Succ} : t \rangle$ where $t \in \mathbb{N}$ is the number of times the Tail destructor had been used prior. Up until now, we had not considered sizes that included variables, though it is a necessity for recursive coinductive terms. We say that the term yielded by *allNats* has size $\lambda^{\mathsf{Tail}} \lambda t. \langle \mathsf{Head} : \langle \mathsf{Zero} : 1 \mid \mathsf{Succ} : t \rangle \rangle$. The remainder of this subsection will deal with the mechanisms to infer these sorts of bounds.

Take special note of the fact that the size term for *allNats* does not actually contain a size for the Tail destructor. I.e., it does not contain a record that looks like $\langle \mathsf{Head} : \alpha \mid \mathsf{Tail} : \beta \rangle$ for some sizes α, β: rather, Head is the only destructor which yields a concrete size and so the term will look like $\lambda^{\mathsf{Tail}} \lambda x. \langle \mathsf{Head} : \alpha \rangle$ where α is a size term over the bound variable x. This is true for all recursive coinductive data types: only the non-recursive destructors will yield concrete sizes; recursive destructors may only influence the sizes yielded by non-recursive destructors.

An example of inference of size bounds on recursive coinductive structures is given in figure 5.17, which infers the size bound of the *allNats* function given in figure 3.16. We infer the following size bound:

$$\lambda^{\mathsf{Tail}} \lambda t. \langle \mathsf{Head} : \langle \mathsf{Zero} : 1 \mid \mathsf{Succ} : t \rangle \rangle$$

This aligns with our intuition about the size of an object defined by coinductive recursion. The Head destructor is the only destructor which yields a non-recursive value. The Tail destructor determines the size of the (inductive) value yielded by the Head destructor. For instance, $\mathsf{Head}(\mathit{allNats}())$ yields a value of size $\langle \mathsf{Zero} : 1 \mid \mathsf{Succ} : 0 \rangle$, specifically the numeral $\overline{0}$, while $\mathsf{Tail}(\mathsf{Tail}(\mathsf{Head}(\mathit{allNats}())))$ yields a value of size $\langle \mathsf{Zero} : 1 \mid \mathsf{Succ} : 2 \rangle$, i.e., the numeral $\overline{2}$. It is not possible to give a finite size bound without any free variables on a recursive coinductive object because, in general, there is no bound on the number of recursive destructors that may be applied to it, and each recursive destruction may increase the size of the object.

Recursive destruction
$$\frac{\Xi \| \Gamma \mid \Delta \vdash t : \alpha}{\Xi \| \Gamma \mid \Delta \vdash \mathsf{D}_i\ t : \alpha[\mathsf{D}_i/x/x+1]}$$

Unfold
$$\frac{\Xi \| \Gamma \mid \Delta \vdash \overline{t} : \alpha \quad \Xi \| \Gamma \mid \Delta \vdash \underline{t}_{\mathsf{D}_i,\alpha} : \lambda d_i.\beta_i}{\Xi \| \Gamma \mid \Delta \vdash \mathbf{unfold}\ t : \lambda^{\mathsf{D}_i} \lambda d_i.\alpha + \sum_i \beta_i} \text{ for each recursive destructor } \mathsf{D}_i$$

Unfold function (pass 1)
$$\frac{\Xi \| \Gamma \mid (g : 0), \Delta \vdash \overline{t}^g : \alpha}{\Xi \| \Gamma \mid \Delta \vdash \overline{\lambda^f g.t} : \alpha}$$

Unfold function parameter (pass 1)
$$\frac{\Xi \| \Gamma \mid (x : x), (g : \lambda\beta.0), \Delta \vdash \overline{t}^g : \alpha}{\Xi \| \Gamma \mid (g : \beta), \Delta \vdash \overline{\lambda^p x.t}^g : \alpha}$$

Unfold non-recursive destructor (pass 1)
$$\frac{\Xi \| \Gamma \mid \Delta \vdash t : \alpha}{\Xi \| \Gamma \mid \Delta \vdash \overline{\lambda^{\mathsf{D}} \mathsf{D}_i.t} : \langle \mathsf{D}_i : \alpha \rangle}$$

Unfold recursive destructor (pass 1)
$$\frac{}{\Xi \| \Gamma \mid \Delta \vdash \overline{\lambda^{\mathsf{D}} \mathsf{D}_i.t} : 0}$$

Unfold co-branches (pass 1)
$$\frac{\Xi \| \Gamma \mid \Delta \vdash \overline{t} : \alpha \quad \Xi \| \Gamma \mid \Delta \vdash \overline{u} : \beta}{\Xi \| \Gamma \mid \Delta \vdash \overline{t \oplus u}^g : \langle \alpha \mid \beta \rangle}$$

Unfold function (pass 2)
$$\frac{\Xi \| \Gamma \mid (g : \beta), \Delta \vdash \underline{t}_{\mathsf{D}_i} : \alpha}{\Xi \| \Gamma \mid \Delta \vdash \underline{\lambda^f g.t}_{\mathsf{D}_i,\beta} : \alpha}$$

Unfold function parameter (pass 2)
$$\frac{\Xi \| \Gamma \mid (x : 0), (g : \beta)\Delta \vdash \underline{t}_{\mathsf{D}_i} : \alpha}{\Xi \| \Gamma \mid (g : \beta), \Delta \vdash \underline{\lambda^p x.t}_{\mathsf{D}_i} : \lambda x.\alpha}$$

Unfold non-matching destructor (pass 2)
$$\frac{}{\Xi \| \Gamma \mid \Delta \vdash \underline{\lambda^{\mathsf{D}} \mathsf{D}_j.t}_{\mathsf{D}_i} : 0}\ \mathsf{D}_i \neq \mathsf{D}_j$$

Unfold matching destructor (pass 2)
$$\frac{\Xi \| \Gamma \mid \Delta \vdash t : \alpha}{\Xi \| \Gamma \mid \Delta \vdash \underline{\lambda^{\mathsf{D}} \mathsf{D}_i.t}_{\mathsf{D}_i} : \lambda d.\alpha}$$

Unfold co-branches (pass 2)
$$\frac{\Xi \| \Gamma \mid \Delta \vdash \underline{t}_{\mathsf{D}_i} : \alpha \quad \Xi \| \Gamma \mid \Delta \vdash \underline{u}_{\mathsf{D}_i} : 0}{\Xi \| \Gamma \mid \Delta \vdash \underline{t \oplus u}_{\mathsf{D}_i} : \alpha}$$

Figure 5.14: Rules of size inference for recursive coinductive size bounds.

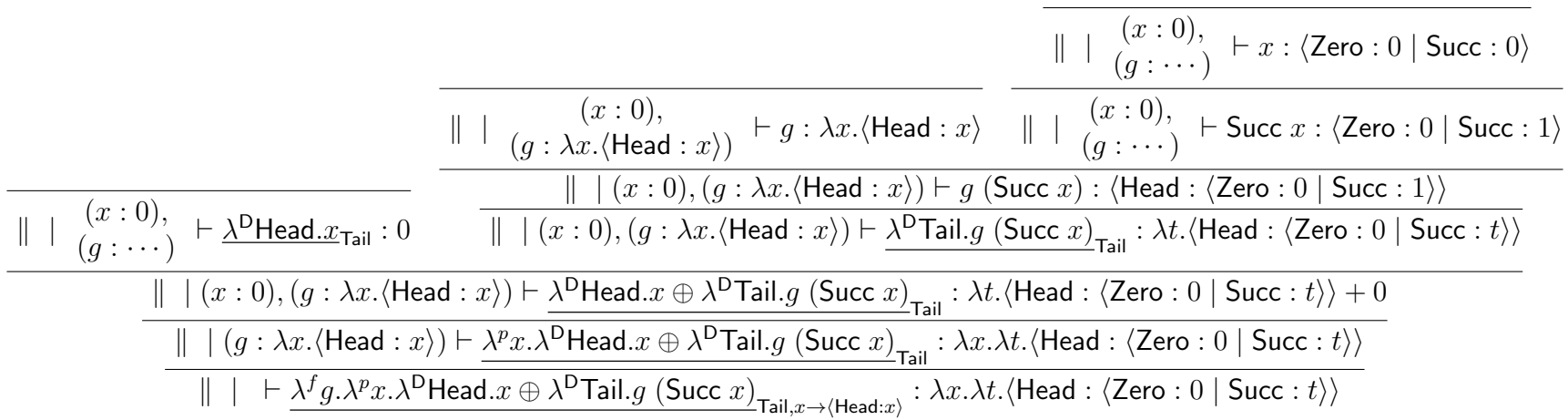

Figure 5.15: A continuation of the derivation given in figure 5.17.

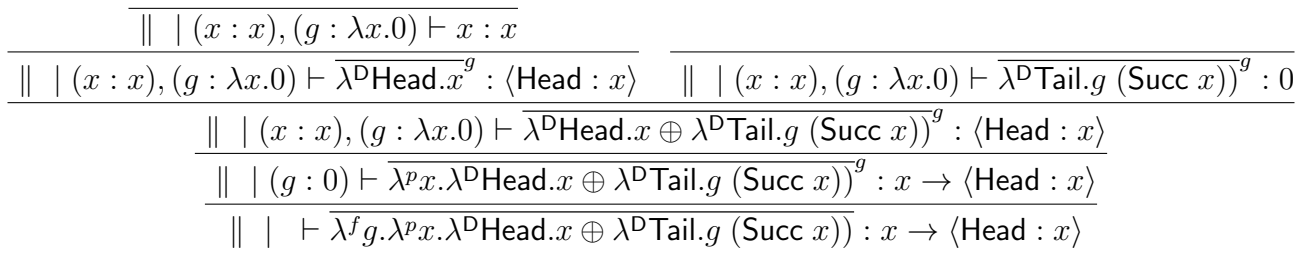

Figure 5.16: A continuation of the derivation below.

Continued in figure 5.16 Continued in figure 5.15

$$\dfrac{\dfrac{}{\| \mid \vdash \mathsf{Zero} : \langle \mathsf{Zero} : 1 \mid \mathsf{Succ} : 0 \rangle} \quad \| \mid \vdash (\lambda^f g.\lambda^p x.\lambda^{\mathsf{D}}\mathsf{Head}.x \oplus \lambda^{\mathsf{D}}\mathsf{Tail}.g\ (\mathsf{Succ}\ x)) : \lambda y.\lambda^{\mathsf{Tail}} t.\langle \mathsf{Head} : \langle \mathsf{Zero} : y.\mathsf{Zero} \mid \mathsf{Succ} : y.\mathsf{Succ} + t \rangle \rangle}{\| \mid \vdash (\mathbf{unfold}\ (\lambda^f g.\lambda^p x.\lambda^{\mathsf{D}}\mathsf{Head}.x \oplus \lambda^{\mathsf{D}}\mathsf{Tail}.g\ (\mathsf{Succ}\ x)))\ \mathsf{Zero} : \lambda^{\mathsf{Tail}} t.\langle \mathsf{Head} : \langle \mathsf{Zero} : 1 \mid \mathsf{Succ} : t \rangle \rangle}\ \mathbf{unfold}$$

Figure 5.17: Size inference on the *allNats* function.

5.1.3 Correctness of inference

To show that size bounds correctly infer sizes, we need a notion of what the correct sizes are. There are two competing concerns when considering size bounds in Compositional Pola. We are interested in the amount of computer memory required to store a value in the programming language, though this will be a secondary concern in this thesis. Of primary concern is the ability to determine how many recursions a data structure may yield if it is used to drive a recursion via the **fold** construct.

When considering the amount of computer memory required to store a value, we will abstract over any computer architecture details such as machine word size, references or pointers to substructures or other structures, and shared memory between data structures. We will assume a computer architecture with uniform memory addressing, but beyond that no specifics. Atomic values in the language are all given a size bound of 1, indicating that they are of constant size. In a real implementation of the language, the sizes of these objects will not all be the same, though they can still be bounded by above from some constant, which is what we designate as 1 for our purposes.

Of special note is the coinductive values, which are not given any concrete size at all in this section. In all practicality, representing a coinductive value in an implementation can be thought of in two ways: (1) as a "multi-closure", i.e., a collection of functions all sharing the same bound free variables; or (2) as an object in the object-oriented sense, a collection of methods all sharing the same instance variables. In this section we do not concern ourselves with the memory required to store executable code, though the reader may assume that the real size required to store a coinductive value is $\mathcal{O}(n+m)$ where n is the size of the free variables and m is the aggregate size of the syntax tree representing the stored destructors.

Defining the potentially recursive size of a term, $\overset{\square}{\bowtie}$

For determining how many recursions a data structure might yield, we will give a formal definition of what we consider to be correct size.

Definition 5. *In the context of environments Ξ, Γ, Δ, for any term t of inductive type $\mathcal{T}$, without loss of generality, say $\mathcal{T}$ has recursive constructors $\mathsf{C}_1, \ldots, \mathsf{C}_n$ and non-recursive constructors $\mathsf{B}_1, \ldots, \mathsf{B}_m$. Further, $t \to t'$ (t evaluates to the term t') under the operational semantics, wherein each recursive constructor C_i having k_i recursive subterms and l_i non-recursive subterms and each non-recursive constructor B_i having p_i subterms, without loss of generality, all recursive subterms come before all non-recursive subterms, we say that t* has inductive size *a, denoted $t \overset{\square\uparrow}{\bowtie} a$, if and only if the following term takes time b under the operation semantics and $\sum_{i=1}^{n} a.\mathsf{C}_i + \sum_{i_1}^{m} a.\mathsf{B}_i \geq \frac{b-3}{3}$:*

$$\begin{gathered}
\mathbf{fold}\,(\lambda^f f.\lambda^p a.\lambda^p b.\mathbf{fcase}\ a \\
(\lambda^{\mathsf{C}}\mathsf{C}_1.\Diamond.\lambda^{op}-r_1.\ldots.\Diamond.\lambda^{op}-r_{k_1}.\Diamond.\lambda^p s_1.\ldots.\Diamond.\lambda^p s_{l_1}.\nabla.f\ r_1 \cdots (f\ r_{k-2}\ (f\ r_{k-1}\ ()))) \\
+\ \cdots \\
+\ \lambda^{\mathsf{C}}\mathsf{C}_n.\Diamond.\lambda^{op}-r_1.\ldots.\Diamond.\lambda^{op}-r_{k_n}.\Diamond.\lambda^p s_1.\ldots.\Diamond.\lambda^p s_{l_n}.\nabla.f\ r_1 \cdots (f\ r_{k-2}\ (f\ r_{k-1}\ ())) \\
+\ \lambda^{\mathsf{C}}\mathsf{B}_1.\Diamond.\lambda^p s_1.\ldots.\Diamond\lambda^p s_{p_1}.() + \cdots + \lambda^{\mathsf{C}}\mathsf{B}_m.\Diamond.\lambda^p s_1.\ldots.\Diamond\lambda^p s_{p_m}.b)) \\
(t)_{\mathbf{o}}\ ()
\end{gathered}$$

The **fold** term in definition 5 indicates a term which does nothing but recurse on all recursive subvalues. We subtract 3 time units from the value b to negate the effects of evaluating t, calling function f initially and evaluating the **fold** initially. We divide b by 3 to negate the effects of constructing the unit () and to negate the effects of evaluating an **fcase** through each recursion.

The motivation behind definition 5 is to define the size of a term as the number of recursions that it can force via a **fold**. This follows from the foundation that the **fold** is the only construct which allows (bounded) recursive computation.

For terms without inductive size, we still need to ensure that the sizes inferred are meaningful and correct, which leads to the next definition.

Definition 6. *In the context of environments Ξ, Γ, Δ, for any term t of coinductive type and where $t \to \langle \mathsf{D}_1 : \alpha_1 \mid \cdots \mid \mathsf{D}_n : \alpha_n \rangle$ under the operational semantics, we say that t* has coinductive size *a, denoted $t \overset{\downarrow\Box}{\bowtie} a$, if and only if, for all $1 \leq i \leq n$, $\alpha_i \overset{\Box}{\bowtie} a$, where the notation $\alpha \overset{\Box}{\bowtie} a$ is given in definition 9.*

In other words, coinductive sizes represent sizes which can either immediately be destructed into inductive sizes or which can be destructed into coinductive sizes. Ultimately, any coinductive size must be able to yield an inductive size to have any utility in the context of Compositional Pola.

There is one type of size that still needs to be related to how it can force a **fold**, which is the tuple size, given in the following definition.

Definition 7. *In the context of environments Ξ, Γ, Δ, for any term t of tuple type and where $t \to \alpha_1 \times \cdots \times \alpha_n \times ()$ under the operational semantics, we say that t* has tuple size *a, denoted $t \overset{()\Box()}{\bowtie} a$, if and only if, for all $1 \leq i \leq n$, $\alpha_i \overset{\Box}{\bowtie} a$, where the notation $\alpha \overset{\Box}{\bowtie} a$ is given in definition 9.*

The definition for arrow sizes follows in a similar vein.

Definition 8. *In the context of environments Ξ, Γ, Δ, for any term t of arrow type and where $t \to (x \to \alpha)$ under the operational semantics, we say that t* has arrow size *$x \to a$, denoted $t \overset{\Box\to\Box}{\bowtie} (x \to a)$, if and only if $\alpha \overset{\Box}{\bowtie} a$, where the notation $\alpha \overset{\Box}{\bowtie} a$ is given in definition 9.*

From definition 5, definition 6, definition 7 and definition 8, we define the following notation for the *potentially recursive size* of a term, which we will use to prove the correctness of size bounds.

Definition 9. *In the context of environments Ξ, Γ, Δ, a Compositional Pola term t is said to have* potentially recursive size *a, denoted $t \overset{\Box}{\bowtie} a$, if and only if one of the following four conditions holds:*

1. $t \overset{\Box\uparrow}{\bowtie} a$;
2. $t \overset{\downarrow\Box}{\bowtie} a$;

3. $t \overset{()\square()}{\bowtie} a$*; or*

4. $t \overset{\square\to\square}{\bowtie} a$*.*

Definitions of inferred size bounds, $\overset{\star}{\bowtie}$

Next we introduce the definition of a size as it corresponds to the size bounds inference process described in this chapter, so that we can relate it to the ideal definition of size given in definition 9.

Definition 10. *For environments* Ξ, Γ, Δ *and Compositional Pola term,* t*, we say* t infers inductive size bound a*, denoted* $t \overset{\star\uparrow}{\bowtie} a$*, if and only if* $\Xi \| \Gamma \mid \Delta \vdash t : \langle \cdots \mid \mathsf{C}_i : x_i, y_{i,1}, \ldots, y_{i,k_i} \mid \cdots \rangle$ *and* $a = \sum_i x_i$ *under the rules of size bound inference given in this chapter.*

For example, the Pola term $t = [3, 8, 2]$ will have inferred size $\langle \mathsf{Nil} : 1 \mid \mathsf{Cons} : 3, \langle \mathsf{Zero} : 1 \mid \mathsf{Succ} : 8 \rangle \rangle$. In this case, $1 + 3 = 4$ and so $t \overset{\star\uparrow}{\bowtie} 4$.

We introduce, also, the concept of a relationship between a coinductive size and our notion of size relating to bounding a recursion.

Definition 11. *For environments* Ξ, Γ, Δ *and Compositional Pola term,* t*, we say* t infers coinductive size bound a*, denoted* $t \overset{\downarrow\star}{\bowtie} a$*, if and only if* $\Xi \| \Gamma \mid \Delta \vdash t : \langle \cdots \mid \mathsf{D}_i : \alpha_i \mid \cdots \rangle$ *and, for all* $1 \leq i \leq n$*,* $\alpha_i \overset{\star}{\bowtie} a_i$ *and* $\max_{i=1}^{n} a_i = a$*. The notation* $\alpha_i \overset{\star}{\bowtie} a_i$ *is given in definition 14.*

The same can be done for tuple sizes.

Definition 12. *For environments* Ξ, Γ, Δ *and Compositional Pola term,* t*, we say* t infers tuple size bound a*, denoted* $t \overset{()\star()}{\bowtie} a$*, if and only if* $\Xi \| \Gamma \mid \Delta \vdash t : \alpha_1 \times \cdots \times \alpha_n \times ()$ *and, for all* $1 \leq i \leq n$*,* $\alpha_i \overset{\star}{\bowtie} a_i$ *and* $\max_{i=1}^{n} a_i = a$*. The notation* $\alpha_i \overset{\star}{\bowtie} a_i$ *is given in definition 14.*

Similarly for arrow sizes.

Definition 13. *For environments* Ξ, Γ, Δ *and Compositional Pola term,* t*, we say* t infers arrow size bound a*, denoted* $t \overset{\star\to\star}{\bowtie} a$*, if and only if* $\Xi \| \Gamma \mid \Delta \vdash t : (x \to \alpha)$ *and* $\alpha \overset{\star}{\bowtie} a$*. The notation* $\alpha_i \overset{\star}{\bowtie} a_i$ *is given in definition 14.*

Similarly to how a consolidated metric was defined in section 5.1.3, we can give a consolidated metric for inferred size bounds.

Definition 14. *For environments* Ξ, Γ, Δ *and Compositional Pola term,* t*, we say* t infers size bound a*, denoted* $t \overset{\star}{\bowtie} a$*, if and only if one of the following four conditions holds:*

1. $t \overset{\star\uparrow}{\bowtie} a$*; or*

2. $t \overset{\downarrow\star}{\bowtie} a$*; or*

Notation	Description	Reference
$t \overset{\square\uparrow}{\bowtie} a$	t has inductive size a (can force a recursions in a **fold**)	Definition 5
$t \overset{\downarrow\square}{\bowtie} a$	t has coinductive size a (can have inductive size a after destruction)	Definition 6
$t \overset{()\square()}{\bowtie} a$	t has tuple size a	Definition 7
$t \overset{\square\to\square}{\bowtie} (x \to a)$	t has arrow size a	Definition 8
$t \overset{\square}{\bowtie} a$	t has potentially recursive size a (one of the above 4 definitions)	Definition 9
$t \overset{\star\uparrow}{\bowtie} a$	t has inferred inductive size a	Definition 10
$t \overset{\downarrow\star}{\bowtie} a$	t has potential inferred size a	Definition 11
$t \overset{()\star()}{\bowtie} a$	t has inferred tuple size a	Definition 12
$t \overset{\star\to\star}{\bowtie} (x \to a)$	t has inferred arrow size a	Definition 13
$t \overset{\star}{\bowtie} a$	t has inferred size a (one of the above 4 definitions)	Definition 14

Figure 5.18: A brief summary of each of the 10 definitions introduced in this section.

3. $t \overset{()\star()}{\bowtie} a$; *or*

4. $t \overset{\star\to\star}{\bowtie} a$.

The above 10 bowtie definitions are summarized in a "cheat sheet" seen in figure 5.18.

Relationship between potential recursive size and inferred size bound

Before we move on to the main proof of this section, we need to introduce a bound on the number of recursions within a **fold** construct, as it is vital to all the bounds given in this thesis.

Definition 15. *For a term t of inductive type $\mathcal{T}$, we say that t has n* immediate recursive constructors*, denoted $c_{\mathcal{T}}(t) = n$, if $t \equiv \mathsf{C}_i(x_1, \ldots, x_{n-1}, y_1, \ldots, y_m)$ for some constructor C_i if, without loss of generality, constructor C_i has $n-1$ recursive attributes and m non-recursive attributes.*

Definition 16. *For a term t of inductive type $\mathcal{T}$, we say that t has n* total recursive constructors*, denoted $c^*_{\mathcal{T}}(t) = n$, if one of the following two conditions holds:*

- $c_{\mathcal{T}}(t) = 1$ *and* $n = 1$; *or*
- $t \equiv \mathsf{C}_i(x_1, \ldots, x_p, y_1, \ldots, y_m)$ *and* $\sum_{i=1}^{p} c^*_{\mathcal{T}}(x_i) = n - 1$.

Lemma 1. $t \overset{\square\uparrow}{\bowtie} n$ *if and only if* $c^*_{\mathcal{T}}(t) = n$.

Proof.
- Assume $t \overset{\square\uparrow}{\bowtie} n$. Thus, the time it takes to evaluate the **fold** construct in definition 5 is $3n+3$ time units and consequently the number of recursive function calls is n. As the **fold** construct performs a recursive function call on every total recursive constructor, $c^*_{\mathcal{T}}(t) = n$.
- Assume $c^*_{\mathcal{T}}(t) = n$. Consquently, the number of total recursive constructors is n. Applying the term to the **fold** construct given in definition 5 will perform n recursive calls, consuming $3n+3$ time units and hence $t \overset{\square\uparrow}{\bowtie} n$.

□

Lemma 2. *By the operational semantics given in section 3.5, there may only be at most one invocation of the recursive function f per constructor of the term being folded over.*

Proof. Because the body of a **fold** is an **fcase**, it suffices to show that each branch in the body of the **fcase** has at most 1 invocation of f. A branch is a constructor match, matching on constructor C_i, followed by the introduction of variables for constructor C_i, followed by the term to be evaluated. Without loss of generality, we say constructor C_i has n recursive variables and m non-recursive variables. A branch is then of the form $\lambda^{\mathsf{C}}\mathsf{C}_i.\diamond.\lambda^{op}x_{1,1}x_{1,2}.\diamond.\ldots.\lambda^{op}x_{n,1}x_{n,2}.\diamond.\lambda^{p}y_1.\ldots.\diamond\lambda^{p}y_m.\triangledown.t$ for some term t. Because of the typing restrictions on f, only the variables $x_{i,2}$ for $1 \leq i \leq n$ can be used to cause an invocation of f. Because $x_{i,2}$ for $1 \leq i \leq n$ is a player variable, it may not appear twice in the same term. Consequently, the number of invocations of f for the branch is at most n.

By lemma 1, then, the total number of invocations of recursive function f is at most $c^*_{\mathcal{T}}(n)$. □

A consequence of lemma 2 is that, no matter how the body of a **fold** is constructed, the number of invocations of the recursive function f is bounded by the number of constructors in the term being folded over.

Given the definitions in section 5.1.3 and section 5.1.3, we can prove the relationship between them, which is sufficient to prove that the size bound inference process is correct.

Proposition 2. *If $t \overset{\star}{\bowtie} a$ and $t \overset{\square}{\bowtie} b$, then $a \geq b$.*

Proof. This proof is by induction on the structure of t. The following bullet points comprise all possible cases for the form of t:

- $t \equiv x$ for some variable x. If x is in scope with size a, then $x \overset{\square}{\bowtie} a$ and $x \overset{\star}{\bowtie} a$.
- $t \equiv \mathsf{C}_i\ u$ for some constructor C_i and some term u. By the induction hypothesis, we can say $u \overset{\star}{\bowtie} a_u$ and $u \overset{\square}{\bowtie} b_u$ where $a_u \geq b_u$. It must be that the size was inferred by the Construction rule in figure 5.2 and so $a = a + 1$. From the Construction rule given in figure 3.31, we can see that $b = b_u + 1$. Therefore, $a \geq b$.
- $t \equiv u\ (v)_{\mathbf{o}}$ or $t \equiv u\ v$. u necessarily has arrow type and we can say that $u \overset{\star\to\star}{\bowtie} (x \to a_u)$ and $u \overset{\square\to\square}{\bowtie} (x \to b_u$ where $a_u \geq b_u$ by the induction hypothesis. Further, we can

see that $v \overset{\star}{\bowtie} a_v$ and $v \overset{\square}{\bowtie} b_v$ where $a_u \geq b_u$. It follows by the Opponent application and Player application rules given in figure 5.2 and Opponent application and Player application rules given in figure 3.31 that $(a_u\ a_v) \geq (b_u\ b_v)$.

- $t \equiv ()$. $t \overset{()\star()}{\bowtie} 0$ and $t \overset{()\square()}{\bowtie} 0$. Therefore, $a = b$.

- $t \equiv u \times v$. By the induction hypothesis, $u \overset{()\star()}{\bowtie} a_u$, $u \overset{()\square()}{\bowtie} b_u$, $v \overset{()\star()}{\bowtie} a_v$ and $v \overset{()\square()}{\bowtie} b_v$ where $a_u \geq b_u$ and $a_v \geq b_v$. $a = a_u + a_v$ and $b = b_u + b_v$. Therefore, $a \geq b$.

- $t \equiv \lambda^o x.u$, $t \equiv \lambda^p x.u$, $t \equiv \lambda^f x.u$ or $t \equiv \lambda^{op} x_1 x_2.u$. By the induction hypothesis, $u \overset{\star}{\bowtie} a_u$ and $a \overset{\square}{\bowtie} b_u$ where $a_u \geq b_u$. $t \overset{\star\to\star}{\bowtie} (x \to a_u)$ and $t \overset{\square\to\square}{\bowtie} (x \to b_u)$ and consequently $a \geq b$.

- $t \equiv \textbf{peek}\ r$, $t \equiv \textbf{case}\ r$ or $t \equiv \textbf{fcase}\ x\ r$. In this case, $t \overset{\star\to\star}{\bowtie} (x \to a_r)$ and $t \overset{\square\to\square}{\bowtie} (x \to b_r)$ where $a_r \geq b_r$, by the induction hypothesis. Necessarily, the term r evaluates to a composition of n branches, having inferred sizes $a_1, \ldots, a_n$ and potential recursive sizes $b_1, \ldots, b_n$. By the Constructor match and Construct non-match rules given in figure 3.31, we see that t evaluates to one of the branches. Hence $b = (x \to b_i)$ for some $1 \leq i \leq n$. Since $\forall 1 \leq i \leq n$, $a_i \geq b_i$, it must be that $\max_{i=1}^{n} a_i \geq b$ and hence $a \geq b$.

- $t \equiv \triangledown.u$, $t \equiv \diamond.u$, $t \equiv \lambda_i^{\mathsf{C}}.u$, $t \equiv \lambda \mathsf{D}_i.u$. From the Unit tuple match rule of figure 3.31 and the Unit match rule of figure 5.3, and by the induction hypothesis, we see that $a = a_u + 1$ and $b = b_u + 1$ where $a \geq b$ and hence $a \geq b$.

- $t \equiv \textbf{fold}\ (\lambda^f f.r)\ x$. Here $t \overset{\star}{\bowtie} (x \to \alpha \cdot \delta + \beta)$ where $x \overset{\star}{\bowtie} \alpha$, δ is the size increase inferred in r per invocation of the recursive function f and β is the size of r independent of any invocations of f. By lemma 2, the number of invocations of f by the operation semantics is bound by α. Consequently, where $r \overset{\square}{\bowtie} b_r$, we know that $\alpha \cdot \delta + \beta \geq b_r$ and hence $a \geq b$.

- $t \equiv \mathsf{D}_i\ u$. In this case, it must be that $u \overset{\star\downarrow}{\bowtie} \langle \mathsf{D}_1 : a_1 \mid \cdots \mid \mathsf{D}_n : a_n \rangle$ and $u \overset{\square\downarrow}{\bowtie} \langle \mathsf{D}_1 : b_1 \mid \cdots \mid \mathsf{D}_n : b_n \rangle$ where $\forall 1 \leq i \leq n$, $a_i \geq b_i$. By the Destruction match and Destruction non-match rules in figure 3.32, we see that the destruction yields one of the subterms of u and so $t \overset{\square}{\bowtie} b_i$. Since $a_i \geq b_i$ by the induction hypothesis, the Destruction rule given in figure 5.12 shows that $a \geq b$.

- $t \equiv u \oplus v$. $u \overset{\star\downarrow}{\bowtie} a_u$ and $u \overset{\square\downarrow}{\bowtie} b_u$ where $a_u \geq b_u$. Similarly, $v \overset{\star\downarrow}{\bowtie} a_v$ and $v \overset{\star\downarrow}{\bowtie} b_v$ where $a_v \geq b_v$. By the Destruction match and Destruction non-match rules in figure 3.32, destruction of this term must yield a size of either b_u or b_v. Since $a_u \geq b_u$ and $a_v \geq b_v$, $\max(a_u, a_v) \geq b_u$ and $\max(a_u, a_v) \geq b_v$, as per the Co-branch composition rule in figure 5.12. Since $t \overset{\star\downarrow}{\bowtie} \max(a_u, b_u)$, $a \geq b$.

- $t \equiv$ **unfold** $(\lambda^f g.u)$. In this case, the term u is a set of cobranches. By the **Unfold** rule in figure 5.14, it must be that $t \overset{\downarrow\star}{\bowtie} t_m$ where, for each cobranch u_i in u, $u_i \overset{\star}{\bowtie} z_i$ and $\forall i, z_i \leq t_m$. Further, by the definition given in figure 5.14, $t_m = \alpha + \sum_i \beta_i$ for some upfront cost α and some aggregate potential cost $\sum_i \beta_i$. We can see from the "Pass 2" rules in figure 5.14 that every β_i is 0 except for one matching cobranch, which we will designate β_{match}.

  $t \overset{\square\to\square}{\bowtie} (x \to r)$. Without loss of generality, we will say $r = r_{upfront} + r_{potential}$. We can see from figure 3.32 that the operational semantics will evaluate u. When applied to an argument, the term u, which consists of a series of cobranches, will perform the "Destruction non-match" rules into a "Destruction match" rule is found. This mimics the "pass 1" rules from figure 5.14 and consequently we can say that $r_{upfront} = \alpha$. The cost of executing the actual destruction is given by evaluating the inner term, as can be seen by the "Destruction non-match" rule of figure 3.32. This mirrors the "Unfold matchind destructor (pass 2)" rule of figure 5.14. Thus, by the induction hypothesis, we can assume that $r_{potential} \leq \beta_{match}$. It follows then that $a \geq b$.

- $t \equiv$ **record** u. This follows as the case for the **unfold**, but with the "Record" rule from figure 5.12.

□

And so, the potential recursive size of a term bounds the number of recursions which may take place as a consequence of using that term in a computation.

5.2 Times

The size bound inference described in section 5.1.2 can be useful information in and of itself, as a bound of the size of the term produced by a computation also gives a bound on the maximum value that can be produced, as well. However, calculating a size bound has another very useful property which is that it is necessary for calculating time bounds in general. The computation time required to execute a Compositional Pola program is generally determined by the use of **fold** constructs and there the number of iterations is set by the size of the term that is being folded over.

The time required to execute Compositional Pola programs is, for most operations, a constant, which we will always denote as 1 in this thesis. When inferring practical bounds, it would be necessary to consider different constants for each operation depending on the time required to perform an operation on a particular computer architecture. The interesting operations—those that do not have constant time bounds—are function applications (and, similarly, destructions) and **fold** constructs. The **fold** construct alone provides the most significant source of computational power in Compositional Pola and it is this that will require attention when computing time bounds for **fold** constructs.

A time bound sequent is of the form $\Xi \| \Gamma \mid \Delta \succcurlyeq t \doteq \alpha$ where Ξ, Γ, Δ are mappings of symbols to *size* bounds, t is a Compositional Pola term and α is a time, defined in section 5.2.1.

Note that many rules of time inference necessarily require the clause to infer the size of a term or subterm, though the reverse is not true. Sequents of the form $\Xi \| \Gamma \mid \Delta \vdash t : \alpha$ will be inferring sizes, as described in section 5.1.2. Rules described in section 5.2.5 require the usage of potential time inference, which is denoted as $\Xi \| \Gamma \mid \Delta \Vdash t \doteq \alpha$ where α is a time.

5.2.1 Times

A time is generally a polynomial over the variable sizes, as defined in section 5.1, i.e., over the variables of the form $v, v.\mathsf{C}_i, v.\mathbb{N}$ where v is a symbol in the contexts Ξ, Γ, Δ, $v.\mathsf{C}_i$ is a constructor count of a symbol and $v.\mathbb{N}$ is a tuple projection of a symbol. The context Ξ maps symbols to both time and size bounds (typically lambda times), but Γ, Δ map symbols to only size bounds. Lambda times are also required when considering time bound inference compositionally, though a well-typed Compositional Pola program should never yield a lambda time bound ultimately. Lambda times will be denoted as $\lambda^T x.\Omega$ where x is a bound variable representing a **size** and Ω is itself a time bound. Application is between a lambda and a size term and is carried out as expected.

A time bound, Ω, is one of the following:

Polynomial	$\mathbb{N}[V]$
Lambda	$\lambda^T x.\Omega$
Application	$(\Omega\ V)$
Constructor pattern	$\lambda^{\mathsf{C}}\mathsf{C}_i.\Omega$
Branch composition	$\langle \Omega \mid \Omega \rangle$
Destructor pattern	$\lambda^{\mathsf{D}}\mathsf{D}_i.\Omega$
Co-branch composition	$\Omega \oplus \Omega$

Note that V is the set of variable sizes, as described above.

Operations on times

As with size bounds, we need to be able to perform some arithmetic operations on time bounds within the sequents. Maximum is defined as in section 5.1.1, with maximums of polynomials taken term-wise and maximums of other bounds done by subsize over the structure of the bound and further define the notion of a maximum across branch composition times bounds:

$$\begin{aligned} \max(\langle \Omega_1 \mid \Omega_2 \rangle) &= \max(\Omega_1, \max(\Omega_2)) \\ \max(\Omega) &= \Omega \qquad \text{where } \Omega \text{ is not of the form } \langle \Omega_1 \mid \Omega_2 \rangle \end{aligned}$$

For example:

$$\begin{aligned} & \max(\langle 3 \mid x^2 + 2 \mid x \rangle) \\ = \; & \max(3, \max(\langle x^2 + 2 \mid x \rangle)) \\ = \; & \max(3, \max(x^2 + 2, \max(x))) \\ = \; & \max(3, \max(x^2 + 2, x)) \\ = \; & \max(3, x^2 + 2) \\ = \; & x^2 + 3 \end{aligned}$$

Multiplication between time bounds and size bounds needs to be defined in order to define bounds inference on the **fcase** rule. Where α is a time bound and β is a size bound, we define $\alpha \cdot \beta$ as:

$$\begin{aligned} (\lambda^{\mathsf{C}} \mathsf{C}_i.\delta) \cdot \beta &= \delta \cdot \beta.\mathsf{C}_i \\ (\delta + \gamma) \cdot \beta &= \delta \cdot \beta + \gamma \cdot \beta \end{aligned}$$

These two rules are sufficient to describe time bound inference with **fcase** constructs.

5.2.2 Environments

When inferring time bounds, we require two views of the symbols in the function environment, Ξ. Inferring time bounds is inherently dependent on being able to infer size bounds owing to the fact that the time spent computing a **fold** construct is proportional to the size of the data structure being folded over. When inferring time bounds involving function application, we need to know a time bound of a function with respect to the size of its inputs and we also may need the *size* bound of a function to compute those sizes of inputs.

As an example, consider the Pola term $f(g(\overline{3}))$. If the functions f and g both take parameters in the opponent world and have time bounds as a function of those parameters, then the time bound of the term as a whole is the summation of four parts: (1) the time to construct the value $\overline{3}$; (2) the time to compute the g function; (3) the time to compute the f function; and (4) the constant overhead of function application. However, in order to be able to infer the time to compute the f function, we necessarily need to infer the size of the term produced by the g function.

In our context, Ξ, we will have symbol bindings both of the form $(f : \Phi)$, indicating a size bound, and $(f \doteq \alpha)$, indicating a time bound.

5.2.3 Sequents

As described in section 5.2.2, there are two views of the global environment that need to be considered—$(f : \Phi)$ and $(f \doteq \alpha)$—the size bounds of a function and the time bounds of a function, respectively. Ultimately what we infer is the time bounds of *evaluating* a term, which is a third consideration, and thus we have a mixture of two different sequents. Table 5.1 gives an overview of the two different sequents used in time bound inference. Note that both sequents use the same environments, which allows easy mixing between them.

Sequent use	Sequent form	Result (form of α)
Size bound inference	$\Xi \Vert \Gamma \mid \Delta \vdash t : \alpha$	Size
Time bound inference	$\Xi \Vert \Gamma \mid \Delta \succcurlyeq t \doteq \alpha$	Time

Table 5.1: The three forms of sequents used during time bound inference.

The size bound inference sequents are described in section 5.1.2. Note that a sequent with a conclusion of type size bound lookup will never have a premise that is not a size bound lookup.

Time bound inference sequents will comprise the remainder of this section. It is common for a sequent with time bound inference conclusion to have a premise or multiple premises which are not time bound inferences.

5.2.4 Inductive terms

Figure 5.19 gives time bounds for simple Compositional Pola terms. Inferring time bounds for these cases essentially means determining the time bounds of the subterms and adding them, plus a constant (1). Note that, in the case of player lambda and player application, the size of the argument cannot possibly contribute to the computational time of the function as a player variable cannot drive a recursion.

When computing upper time bounds on branches and associated constructs, we are typically interested in finding the maximum across the branches. Figure 5.20 gives rules of inference for time bounds of these constructs.

Figure 5.24 gives the time bound inference for the *add* function given in figure 3.3, namely a bound of $x.\mathsf{Zero} + 5x.\mathsf{Succ} + 5$. I.e., the time required to compute the *add* function is independent of the variable y (which must be the case because y is a player variable and cannot drive a recursion) and requires time units bounded by the number of Zero constructors in x plus 5 times the number of Succ constructors plus a constant 5 time units. Since it is not possible for a natural number to have a number of Zero constructors other than one, the time bound is actually $5x.\mathsf{Succ} + 6$, or five times the value of x plus 6 time units. Ignoring the coefficients, this aligns with our expectation of what the time required to compute addition in Peano arithmetic should be.

Note that the multiplication operation to correctly compute the affine time bounds of the *add* function is done by the **fcase** rule.

Constructor
$$\frac{}{\Xi \| \Gamma \mid \Delta \succcurlyeq \mathsf{C}_i \doteq 0}$$

Variable reference
$$\frac{}{\Xi \| \Gamma \mid \Delta \succcurlyeq x \doteq 1} \text{ where } x \in (\Gamma \cup \Delta)$$

Opponent application
$$\frac{\Xi \| \Gamma \mid \Delta \succcurlyeq t \doteq \alpha_T \quad \Xi \| \Gamma \mid \Delta \vdash u : \beta_Z \quad \Xi \| \Gamma \mid \Delta \succcurlyeq u \doteq \beta_T}{\Xi \| \Gamma \mid \Delta \succcurlyeq (t\ (u)_{\mathbf{o}}) \doteq \alpha_T[\beta_Z] + \beta_T + 1}$$

Player application
$$\frac{\Xi \| \Gamma \mid \Delta \succcurlyeq t \doteq \alpha \quad \Xi \| \Gamma \mid \Delta \succcurlyeq u \doteq \beta}{\Xi \| \Gamma \mid \Delta \succcurlyeq (t\ u) \doteq \alpha + \beta + 1}$$

Tuple unit
$$\frac{}{\Xi \| \Gamma \mid \Delta \succcurlyeq () \doteq 1}$$

Tuple pair
$$\frac{\Xi \| \Gamma \mid \Delta \succcurlyeq t \doteq \alpha \quad \Xi \| \Gamma \mid \Delta \succcurlyeq u \doteq \beta}{\Xi \| \Gamma \mid \Delta \succcurlyeq t \times u \doteq \alpha + \beta + 1}$$

Opponent lambda
$$\frac{\Xi \| (x : x), \Gamma \mid \Delta \succcurlyeq t \doteq \alpha}{\Xi \| \Gamma \mid \Delta \succcurlyeq \lambda^o x.t \doteq \lambda^T x.\alpha}$$

Player lambda
$$\frac{\Xi \| \Gamma \mid (x : x), \Delta \succcurlyeq t \doteq \alpha}{\Xi \| \Gamma \mid \Delta \succcurlyeq \lambda^p x.t \doteq \alpha}$$

Figure 5.19: Time bounds for simple Compositional Pola terms.

Peek
$$\frac{\Xi\|\Gamma \mid \Delta \succcurlyeq s \doteq \alpha \quad \Xi\|\Gamma \mid \Delta \vdash s : \gamma \quad \Xi\|\Gamma \mid (C : \gamma), \Delta \succcurlyeq b \doteq \beta}{\Xi\|\Gamma \mid \Delta \succcurlyeq \mathbf{peek}\ b\ s \doteq \alpha + \max(\beta)}$$

Case
$$\frac{\Xi\|\Gamma \mid \Delta \succcurlyeq s \doteq \alpha \quad \Xi\|\Gamma \mid \Delta \vdash s : \gamma \quad \Xi\|\Gamma \mid (C : \gamma), \Delta \succcurlyeq b \doteq \beta}{\Xi\|\Gamma \mid \Delta \succcurlyeq \mathbf{case}\ b\ s \doteq \alpha + \max(\beta)}$$

FCase
$$\frac{\Xi\|\Gamma \mid \Delta \vdash x : \alpha_Z \quad \Xi\|\Gamma \mid \Delta \succcurlyeq x \doteq \alpha_T \quad \Xi\|\Gamma \mid \Delta \succcurlyeq y \doteq \beta_T}{\Xi\|\Gamma \mid \Delta \succcurlyeq \mathbf{fcase}\ x\ t \doteq \beta_T \cdot \alpha_Z + \alpha_T}$$

Fold
$$\frac{\Xi\|\Gamma \mid \Delta \succcurlyeq b \doteq \alpha}{\Xi\|\Gamma \mid \Delta \succcurlyeq \mathbf{fold}\ (\lambda^f f.\lambda^p x.b) \doteq \lambda^{\mathsf{T}} x.\alpha}$$

Branch composition
$$\frac{\Xi\|\Gamma \mid \Delta \succcurlyeq t \doteq \alpha \quad \Xi\|\Gamma \mid \Delta \succcurlyeq u \doteq \beta}{\Xi\|\Gamma \mid \Delta \succcurlyeq t + u \doteq \langle \alpha \mid \beta \rangle}$$

Constructor match
$$\frac{\Xi\|\Gamma \mid (C : C.\mathsf{C}_i), (N : 0), \Delta \succcurlyeq t \doteq \alpha}{\Xi\|\Gamma \mid (C : C).\Delta \succcurlyeq \lambda\mathsf{C}.\mathsf{C}_i.t \doteq \lambda_i^{\mathsf{C}}.\alpha}$$

Opponent variable match
$$\frac{\Xi\|\Gamma \mid (C : \alpha_C), (x : \alpha_C)\Delta \succcurlyeq t \doteq \alpha_t}{\Xi\|\Gamma \mid (C : \alpha_C), \Delta \succcurlyeq \Diamond.\lambda^o x.t \doteq \alpha_t}\ x \text{ is recursive}$$

Opponent variable match
$$\frac{\Xi\|(x : \alpha_C.n), \Gamma \mid (C : \alpha_C), (N : n+1), \Delta \succcurlyeq t \doteq \alpha_t}{\Xi\|\Gamma \mid (C : \alpha_C), (N : n), \Delta \succcurlyeq \Diamond.\lambda^o x.t \doteq \alpha_t}\ \text{otherwise}$$

Player variable match
$$\frac{\Xi\|\Gamma \mid (C : \alpha_C), (x : \alpha_C)\Delta \succcurlyeq t \doteq \alpha_t}{\Xi\|\Gamma \mid (C : \alpha_C), \Delta \succcurlyeq \Diamond.\lambda^p x.t \doteq \alpha_t}\ x \text{ is recursive}$$

Player variable match
$$\frac{\Xi\|\Gamma \mid (C : \alpha_C), (N : n+1), (x : \alpha_C.n), \Delta \succcurlyeq t \doteq \alpha_t}{\Xi\|\Gamma \mid (C : \alpha_C), (N : n), \Delta \succcurlyeq \Diamond.\lambda^p x.t \doteq \alpha_t}\ \text{otherwise}$$

Fold variable match
$$\frac{\Xi\|(x : \alpha_C), \Gamma \mid (y : \alpha_C), \Delta \succcurlyeq t \doteq \alpha_t}{\Xi\|\Gamma \mid (C : \alpha_C), \Delta \succcurlyeq \Diamond.\lambda^{op} xy.t \doteq \alpha_t}$$

Unit match
$$\frac{\Xi\|\Gamma \mid \Delta \succcurlyeq t \doteq \alpha_t}{\Xi\|\Gamma \mid \Delta \succcurlyeq \triangledown.t \doteq \alpha_t}$$

Figure 5.20: Time bounds for inductive constructs in Compositional Pola.

$$\dfrac{\dfrac{\overline{\cdots \succcurlyeq f \doteq 0} \quad \overline{\cdots \succcurlyeq z \doteq 1}}{(f:0),(f \doteq 0) \| (x:x) \mid (z:x.\mathsf{Succ}),(y:y) \succcurlyeq f\ z \doteq 2} \quad \dfrac{\overline{\cdots \succcurlyeq \mathsf{Succ} \doteq 0} \quad \overline{\cdots \succcurlyeq y \doteq 1}}{\begin{matrix}(f:0),\\(f \doteq 0)\end{matrix} \| (x:x) \mid \begin{matrix}(z:x.\mathsf{Succ}),\\(y:y)\end{matrix} \succcurlyeq \mathsf{Succ}\ y \doteq 2}}{(f:0),(f \doteq 0) \| (x:x) \mid (z:x.\mathsf{Succ}),(y:y) \succcurlyeq f\ z\ (\mathsf{Succ}\ y) \doteq 5}$$

Figure 5.21: A continuation of the derivation below.

$$\dfrac{\dfrac{\dfrac{\overline{(f:0),(f \doteq 0) \| (x:x) \mid (y:y) \succcurlyeq y \doteq 1}}{\begin{matrix}(f:0),\\(f \doteq 0)\end{matrix} \| (x:x) \mid \begin{matrix}(C:x.\mathsf{Zero}),\\(N:0),(y:y)\end{matrix} \succcurlyeq \triangledown.y \doteq 1} \quad \dfrac{\text{Continued in figure 5.21}}{(f:0),(f \doteq 0) \| (x:x) \mid \begin{matrix}(C:x.\mathsf{Succ}),(N:0),\\(z:x.\mathsf{Succ}),(y:y)\end{matrix} \succcurlyeq \triangledown.f\ z\ (\mathsf{Succ}\ y) \doteq 5}}{\begin{matrix}(f:0),\\(f \doteq 0)\end{matrix} \| (x:x) \mid \begin{matrix}(C:x.\mathsf{Succ}),\\(N:0),(y:y)\end{matrix} \succcurlyeq \Diamond.\lambda^{op} - z.\triangledown.f\ z\ (\mathsf{Succ}\ y) \doteq 5}}{\begin{matrix}(f:0),\\(f \doteq 0)\end{matrix} \| (x:x) \mid \begin{matrix}(C:x),\\(y:y)\end{matrix} \succcurlyeq \lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.y \doteq \langle \mathsf{Zero}:1\rangle \quad \begin{matrix}(f:0),\\(f \doteq 0)\end{matrix} \| (x:x) \mid \begin{matrix}(C:x),\\(y:y)\end{matrix} \succcurlyeq \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} - z.\triangledown.f\ z\ (\mathsf{Succ}\ y) \doteq \langle \mathsf{Succ}:5\rangle}{(f:0),(f \doteq 0) \| (x:x) \mid (C:x),(y:y) \succcurlyeq \lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.y + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} - z.\triangledown.f\ z\ (\mathsf{Succ}\ y) \doteq \langle \mathsf{Zero}:1 \mid \mathsf{Succ}:5\rangle}$$

Figure 5.22: A continuation of the derivation below.

$$\dfrac{\dfrac{\dfrac{\overline{(f:0),(f \doteq 0) \| (x:x) \mid (y:y) \vdash x:x} \quad \overline{(f:0),(f \doteq 0) \| (x:x) \mid (y:y) \succcurlyeq x \doteq 1} \quad \text{Continued in figure 5.22}}{(f:0),(f \doteq 0) \| (x:x) \mid (y:y) \succcurlyeq \mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.y + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} - z.\triangledown.f\ z\ (\mathsf{Succ}\ y)) \doteq x.\mathsf{Zero} + 5x.\mathsf{Succ} + 2}}{(f:0),(f \doteq 0) \| (x:x) \mid (y:y) \succcurlyeq \lambda^{p}y.\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.y + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} - z.\triangledown.f\ z\ (\mathsf{Succ}\ y)) \doteq x.\mathsf{Zero} + 5x.\mathsf{Succ} + 2}}{\| (x:x) \mid (y:y) \succcurlyeq \mathbf{fold}\ (\lambda^{f}f.\lambda^{p}x.\lambda^{p}y.\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.y + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} - z.\triangledown.f\ z\ (\mathsf{Succ}\ y))) \doteq \lambda^{T}x.(x.\mathsf{Zero} + 5x.\mathsf{Succ} + 2)}$$

Figure 5.23: A continuation of the below derivation.

$$\dfrac{\dfrac{\text{Continued in figure 5.23} \quad \overline{\|(x:x) \mid (y:y) \vdash x:x} \quad \overline{\|(x:x) \mid (y:y) \succcurlyeq x \doteq 1}}{\|(x:x) \mid (y:y) \succcurlyeq \mathbf{fold} \begin{pmatrix} \lambda^f f.\lambda^p x.\lambda^p y.\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.y \\ +\ \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} - z.\triangledown.f\ z\ (\mathsf{Succ}\ y)) \end{pmatrix} (x)_{\mathbf{o}} \doteq x.\mathsf{Zero} + 5x.\mathsf{Succ} + 3} \quad \overline{\|(x:x) \mid (y:y) \succcurlyeq y \doteq 1}}{\|(x:x) \mid (y:y) \succcurlyeq \mathbf{fold} \begin{pmatrix} \lambda^f f.\lambda^p x.\lambda^p y.\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.y \\ +\ \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} - z.\triangledown.f\ z\ (\mathsf{Succ}\ y)) \end{pmatrix} (x)_{\mathbf{o}}\ y \doteq x.\mathsf{Zero} + 5x.\mathsf{Succ} + 5}$$

Figure 5.24: Time bound inference for the *add* function given is figure 3.3.

5.2.5 Coinductive terms

Similarly to how bounds inference on coinductive terms was handled when inferring size bounds in section 5.1.2 and section 5.1.2, time bounds with respect to coinductive terms are dealt with by representing the potential of the term. E.g., for a typical record, the up-front time cost is a small constant (1, with respect to the operational semantics), but there is also a potential time bound of what the time cost would be if the term were destructed.

As described in section 5.2, there is a form of sequent, $\Xi \| \Gamma \mid \Delta \Vdash t \doteq \alpha$, which is used to infer potential time bounds. This extra form of sequent is required because inference on a record will necessarily need to infer two time bounds: an upfront—or concrete—bound and a potential bound.

Figure 5.25 gives rules of inference for non-recursive coinductive terms. Here we require two different types of time bound inference: upfront time bounds and potential time bounds. Upfront time bounds are upper bounds on the number of computational steps required, as per the operational semantics, to evaluate the expression. These are generally a very low constant for coinductive objects and do not vary with the complexity of the terms defined within the destructors: because the semantics surrounding coinductive objects corresponding simply to "packing up" the terms to be executed later, there is no substantial computational cost. The potential cost corresponds to the computational cost of destructing the object with a given destructor.

As an example, consider the Pola[1] term $(\ \mathsf{P1} : \overline{3}; \mathsf{P2} : [\,]\ )$, which would have a size of $\langle \mathsf{P1} : \langle \mathsf{Zero} : 1 \mid \mathsf{Succ} : 3 \rangle \mid \mathsf{P2} : \langle \mathsf{Nil} : 1 \mid \mathsf{Cons} : 0, 0 \rangle \rangle$. The upfront time bound on this term is 2, corresponding to the cost to packaging up each destructor. The potential time bound is $\langle \mathsf{P1} : 4 \mid \mathsf{P2} : 1 \rangle$. Consequently, the term $\mathsf{P1}[(\ \mathsf{P1} : \overline{3}; \mathsf{P2} : [\,]\ )]$ has time bound $1+2+4 = 7$, which corresponds exactly to what the cost would be under the operational semantics.

When dealing with recursive coinductive constructs, any extra potential computational cost is carried by the act of destruction, and thus the **unfold** construct per se does not add any additional potential cost.

Note that each rule of inference for inductive constructs given in figures 5.19 and 5.20 needs a corresponding rule for inferring a potential bound, in case a coinductive construct is produced via an inductive construct. Figures 5.26 and 5.27 give potential rules of inference for inductive constructs.

5.2.6 Correctness of inference

Here we show that the type inference rules are correct, in that they provide an upper bound on the running time necessary to compute a particular Compositional Term, under the operational semantics given in section 3.5. The structure of the proof follows in a similar style to the correctness of size inference bounds given in section 5.1.3 with the

[1] In Compositional Pola, this would be written as follows:

$$\textbf{record}\ (\lambda^{\mathsf{D}}\mathsf{P1}.\mathsf{Succ}(\mathsf{Succ}(\mathsf{Succ}((\mathsf{Zero}\ ()) \times ()) \times ()) \times ()) \oplus \lambda^{\mathsf{D}}\mathsf{P2}.\mathsf{Nil}\ ())$$

Destruction
$$\frac{\Xi \| \Gamma \mid \Delta \succcurlyeq t \doteq \beta \quad \Xi \| \Gamma \mid \Delta \Vdash t \doteq \alpha}{\Xi \| \Gamma \mid \Delta \succcurlyeq t\ \mathsf{D}_i \doteq \beta + \alpha[\mathsf{D}_i] + 1}$$

Record (upfront)
$$\frac{\Xi \| \Gamma \mid \Delta \succcurlyeq t \doteq \alpha}{\Xi \| \Gamma \mid \Delta \succcurlyeq \mathbf{record}\ t \doteq \alpha}$$

Record (potential)
$$\frac{\Xi \| \Gamma \mid \Delta \Vdash t \doteq \alpha}{\Xi \| \Gamma \mid \Delta \Vdash \mathbf{record}\ t \doteq \alpha}$$

Co-branch composition (upfront)
$$\frac{\Xi \| \Gamma \mid \Delta \succcurlyeq t \doteq \alpha \quad \Xi \| \Gamma \mid \Delta \succcurlyeq u \doteq \beta}{\Xi \| \Gamma \mid \Delta \succcurlyeq t \oplus u \doteq \alpha + \beta}$$

Co-branch composition (potential)
$$\frac{\Xi \| \Gamma \mid \Delta \Vdash t \doteq \alpha \quad \Xi \| \Gamma \mid \Delta \Vdash u \doteq \beta}{\Xi \| \Gamma \mid \Delta \Vdash t \oplus u \doteq \langle \alpha \mid \beta \rangle}$$

Destructor (upfront)
$$\frac{}{\Xi \| \Gamma \mid \Delta \succcurlyeq \lambda^{\mathsf{D}} \mathsf{D}_i.t \doteq 1}$$

Destructor (potential)
$$\frac{\Xi \| \Gamma \mid \Delta \succcurlyeq t \doteq \alpha}{\Xi \| \Gamma \mid \Delta \Vdash \lambda^{\mathsf{D}} \mathsf{D}_i.t \doteq \langle \mathsf{D}_i.\alpha \rangle}\ \mathsf{D}_i \text{ is not recursive}$$

Unfold (upfront)
$$\frac{\Xi \| \Gamma \mid \Delta \succcurlyeq t \doteq \alpha}{\Xi \| \Gamma \mid \Delta \succcurlyeq \mathbf{unfold}\ (\lambda^f g.t) \doteq \alpha + 1}$$

Unfold (potential)
$$\frac{\Xi \| \Gamma \mid \Delta \Vdash t : \alpha}{\Xi \| \Gamma \mid \Delta \Vdash \mathbf{unfold}\ (\lambda^f g.t) \doteq \alpha}$$

Figure 5.25: Time bounds for coinductive terms.

$$\text{Variable reference} \quad \frac{}{\Xi \| \Gamma \mid \Delta \Vdash x \doteq x} \text{ where } x \in (\Gamma \cup \Delta)$$

$$\text{Opponent application} \quad \frac{\Xi \| \Gamma \mid \Delta \Vdash t \doteq \alpha_T \quad \Xi \| \Gamma \mid \Delta \vdash u : \beta_Z}{\Xi \| \Gamma \mid \Delta \Vdash (t\ (u)_{\mathbf{o}}) \doteq \alpha_T[\beta_Z]}$$

$$\text{Player application} \quad \frac{\Xi \| \Gamma \mid \Delta \Vdash t \doteq \alpha_T \quad \Xi \| \Gamma \mid \Delta \vdash u : \beta_Z}{\Xi \| \Gamma \mid \Delta \Vdash (t\ u) \doteq \alpha_T[\beta_Z]}$$

$$\text{Opponent lambda} \quad \frac{\Xi \| (x : x), \Gamma \mid \Delta \Vdash t \doteq \alpha}{\Xi \| \Gamma \mid \Delta \Vdash \lambda^o x.t \doteq \lambda^T x.\alpha}$$

$$\text{Player lambda} \quad \frac{\Xi \| \Gamma \mid (x : x), \Delta \Vdash t \doteq \alpha}{\Xi \| \Gamma \mid \Delta \succcurlyeq \lambda^p x.t \doteq \lambda^T x.\alpha}$$

Figure 5.26: Potential time bounds for simple Compositional Pola terms.

simplification that there is no question about the best definition to use for considering the time of a term as there was when considering the size of a term.

Definition 17. *In the context of environments Ξ, Γ, Δ, for any term t, we say t* has time *a, denoted $t \overset{\square}{\asymp} a$, if $t \Rightarrow t'$ for some term t' taking precisely a steps under the operational semantics.*

There is one complication when considering time bounds, though, which is that time bound inference has a more complex interplay with potential bounds than size bound inference did. When inferring time bounds of coinductive terms, we infer both a potential size and potential time bound, where *both* may be relevant to the ultimate time bound. Even in the case of inductive terms, the size bound of a term can be required to determine the time bound of a **fold** construct. In this sense we will necessarily assume correctness of any size bounds, as given in section 5.1.3. This interplay will become evident in lemma 3 where we prove the relationship between time bound and inferred time bound.

We introduce now the notion of an inferred time bound, which we will relate to definition 17.

Definition 18. *For environments Ξ, Γ, Δ and Compositional Pola term, t, we say t* infers time bound *a, denoted $t \overset{\star}{\asymp} a$, if and only if $\Xi \| \Gamma \mid \Delta \succcurlyeq t \doteq p$ under the rules of size bound inference given in this chapter for some polynomial p.*

Note that we need no definition for potential time bounds, as we only need to worry about the correspondence between actual time bounds and times at this point.

Lemma 3. *If $t \overset{\star}{\asymp} a$ and $t \overset{\square}{\asymp} b$, then $a \geq b$.*

Proof. This proof is by induction on the structure of t.

Peek
$$\frac{\Xi \| \Gamma \mid \Delta \vdash s : \gamma \quad \Xi \| \Gamma \mid (C : \gamma), \Delta \Vdash b \doteq \beta}{\Xi \| \Gamma \mid \Delta \Vdash \mathbf{peek}\ b\ s \doteq \max(\beta)}$$

Case
$$\frac{\Xi \| \Gamma \mid \Delta \vdash s : \gamma \quad \Xi \| \Gamma \mid (C : \gamma), \Delta \Vdash b \doteq \beta}{\Xi \| \Gamma \mid \Delta \Vdash \mathbf{case}\ b\ s \doteq \max(\beta)}$$

FCase
$$\frac{\Xi \| \Gamma \mid \Delta \vdash x : \alpha_Z \quad \Xi \| \Gamma \mid \Delta \Vdash y \doteq \beta_T}{\Xi \| \Gamma \mid \Delta \Vdash \mathbf{fcase}\ x\ t \doteq \beta_T \cdot \alpha_Z}$$

Fold
$$\frac{\Xi \| \Gamma \mid \Delta \Vdash b \doteq \alpha}{\Xi \| \Gamma \mid \Delta \Vdash \mathbf{fold}\ (\lambda^f f.\lambda^p x.b) \doteq \lambda^{\mathsf{T}} x.\alpha}$$

Branch composition
$$\frac{\Xi \| \Gamma \mid \Delta \Vdash t \doteq \alpha \quad \Xi \| \Gamma \mid \Delta \Vdash u \doteq \beta}{\Xi \| \Gamma \mid \Delta \Vdash t + u \doteq \langle \alpha \mid \beta \rangle}$$

Constructor match
$$\frac{\Xi \| \Gamma \mid (C : C.\mathsf{C}_i), (N : 0), \Delta \Vdash t \doteq \alpha}{\Xi \| \Gamma \mid (C : C).\Delta \Vdash \lambda \mathsf{C}.\mathsf{C}_i.t \doteq \lambda_i^{\mathsf{C}}.\alpha}$$

Opponent variable match
$$\frac{\Xi \| \Gamma \mid (C : \alpha_C), (x : \alpha_C)\Delta \Vdash t \doteq \alpha_t}{\Xi \| \Gamma \mid (C : \alpha_C), \Delta \Vdash \Diamond.\lambda^o x.t \doteq \alpha_t}\ x \text{ is recursive}$$

Opponent variable match
$$\frac{\Xi \| (x : \alpha_C.n), \Gamma \mid (C : \alpha_C), (N : n+1), \Delta \Vdash t \doteq \alpha_t}{\Xi \| \Gamma \mid (C : \alpha_C), (N : n), \Delta \Vdash \Diamond.\lambda^o x.t \doteq \alpha_t}\ \text{otherwise}$$

Player variable match
$$\frac{\Xi \| \Gamma \mid (C : \alpha_C), (x : \alpha_C)\Delta \Vdash t \doteq \alpha_t}{\Xi \| \Gamma \mid (C : \alpha_C), \Delta \Vdash \Diamond.\lambda^p x.t \doteq \alpha_t}\ x \text{ is recursive}$$

Player variable match
$$\frac{\Xi \| \Gamma \mid (C : \alpha_C), (N : n+1), (x : \alpha_C.n), \Delta \Vdash t \doteq \alpha_t}{\Xi \| \Gamma \mid (C : \alpha_C), (N : n), \Delta \Vdash \Diamond.\lambda^p x.t \doteq \alpha_t}\ \text{otherwise}$$

Fold variable match
$$\frac{\Xi \| (x : \alpha_C), \Gamma \mid (y : \alpha_C), \Delta \Vdash t \doteq \alpha_t}{\Xi \| \Gamma \mid (C : \alpha_C), \Delta \Vdash \Diamond.\lambda^{op} xy.t \doteq \alpha_t}$$

Unit match
$$\frac{\Xi \| \Gamma \mid \Delta \Vdash t \doteq \alpha_t}{\Xi \| \Gamma \mid \Delta \Vdash \triangledown.t \doteq \alpha_t}$$

Figure 5.27: Potential time bounds for inductive constructs in Compositional Pola.

- $t \equiv x$ for some variable x, $t \equiv ()$, $t \equiv \lambda^o x.u$, $t \equiv \lambda^p x.u$, $t \equiv \lambda^f x.u$ or $t \equiv \lambda^{op} x_1 x_2.u$. $a = b = 1$.

- $t \equiv \mathsf{C}_i\ u$ for some constructor C_i and some term u. By the induction hypothesis, we can say $u \overset{\star}{\asymp} a_u$ and $u \overset{\square}{\asymp} b_u$ where $a_u \geq b_u$. $b = b_u + 1$. $a = a_u + 1$. Therefore, $a \geq b$.

- $t \equiv u\ (v)_{\mathbf{o}}$ or $t \equiv u\ v$. u necessarily has arrow type and we can say that u infers *potential* time bound $\lambda^T x.a_u$ and that u evaluates to either $\lambda^p x.z_u$ or $\lambda^o x.z_u$ where and $z_u \overset{\square}{\asymp} b_u$ and $a_u \geq b_u$ by the induction hypothesis. Considering the immediate computational costs of evaluating u before application, we see that $u \overset{\star}{\asymp} x_u$ and $u \overset{\square}{\asymp} y_u$ where $x_u \geq y_u$. Further, we can see that $v \overset{\star}{\asymp} a_v$ and $v \overset{\square}{\asymp} b_v$ where $a_u \geq b_u$. It follows that $(a_u\ a_v) \geq (b_u\ b_v)$. $t \overset{\star}{\asymp} x_u + a_u[a_v] + 1$ and $t \overset{\square}{\asymp} y_u + b_u[b_v] + 1$ and hence $a \geq b$.

- $t \equiv u \times v$. By the induction hypothesis, $u \overset{(\star}{\asymp} a_u$, $u \overset{\square}{\asymp} b_u$, $v \overset{\star}{\asymp} a_v$ and $v \overset{\square}{\asymp} b_v$ where $a_u \geq b_u$ and $a_v \geq b_v$. $a = a_u + a_v + 1$ and $b = b_u + b_v + 1$. Therefore, $a \geq b$.

- $t \equiv \mathbf{peek}\ b\ r$, $t \equiv \mathbf{case}\ b\ r$ or $t \equiv \mathbf{fcase}\ r\ b$. In this case, $r \overset{\star}{\bowtie} a_r$ and $r \overset{\square}{\bowtie} b_r$ where $a_r \geq b_r$. By the induction hypothesis, we also see that $b \overset{\star}{\asymp} a_b$ and $b \overset{\square}{\asymp} b_b$ where $a_b \geq b_b$. It follows that $a_b[a_r] \geq b_b[b_r]$. Necessarily, the term b evaluations to a composition of n branches, having inferred times $a_1, \ldots, a_n$ and actual times $b_1, \ldots, b_n$. By the operational semantics, t evaluates to one of the branches and hence $b_b[b_r] = b_i$ for some $1 \leq i \leq n$. Since $\forall 1 \leq i \leq n$, $a_i \geq b_i$, it must be that $\max_{i=1} n a_i \geq b$. It follows, then, that $a \geq b$.

- $t \equiv \nabla.u$, $t \equiv \diamond.u$, $t \equiv \lambda^{\mathsf{C}}_i.u$, $t \equiv \lambda \mathsf{D}_i.u$. This follows immediately from the induction hypothesis.

- $t \equiv \mathbf{fold}\ (\lambda^f f.r)\ x$. Here $(\mathbf{fold}\ (\lambda^f f.r)) \overset{\star}{\asymp} \lambda^{\mathsf{T}} x.a_r$ where $r \overset{\star}{\asymp} a_r$ and $r \overset{\square}{\asymp} b_r$ where $a_r \geq b_r$. Due to lemma 2, the number of calls to f is bounded by z_x where $x \overset{\star}{\bowtie} z_x$ and $x \overset{\square}{\bowtie} y_x$ where $z_x \geq y_x$. Note also that $x \overset{\star}{\asymp} a_x$ and $x \overset{\square}{\asymp} b_x$ where $a_x \geq b_x$. Consequently, it must be that $b \leq b_r \cdot y_x + b_x$. Since $a_r \geq b_r, z_x \geq y_x, a_x \geq b_x$, it must be that $a_r[z_x] + a_x \geq b$ and hence $a \geq b$.

- $t \equiv \mathsf{D}_i\ u$. Because of the rule of time bound inference for destruction, specifically determining $\Xi \| \Gamma \mid \Delta \Vdash u \doteq u_a$, where $u_a \equiv \langle \mathsf{D}_1 : u_{a,1} \mid \cdots \mid \mathsf{D}_n : u_{a,n} \rangle$, we know that, for all destructors $\mathsf{D}_1, \ldots, \mathsf{D}_j, \ldots, \mathsf{D}_n$, $(\mathsf{D}_j\ u) \overset{\square}{\asymp} b_j$ and $a_j \geq b_j$. Consequently, $a_i \geq b_i$ and hence $a \geq b$. In short, the time required to evaluate a destruction is that of the potential time of the record, which is bound correctly via the induction hypothesis.

- $t \equiv u \oplus v$. $u \overset{\star\downarrow}{\asymp} a_u$ and $u \overset{\square\downarrow}{\asymp} b_u$ where $a_u \geq b_u$. Similarly, $v \overset{\star\downarrow}{\asymp} a_v$ and $v \overset{\star\downarrow}{\asymp} b_v$ where $a_v \geq b_v$. By the operational semantics, destruction of this term must yield a time of

either b_u or b_v. Since $a_u \geq b_u$ and $a_v \geq b_v$, $\max(a_u, a_v) \geq b_u$ and $\max(a_u, a_v) \geq b_v$. Since $t \overset{\star\downarrow}{\asymp} \max(a_u, b_u)$, $a \geq b$.

- $t \equiv$ **unfold** $(\lambda^f.r)\ x$ or $t \equiv$ **record** $r\ x$. We consider two aspects to this: the upfront cost (the cost of evaluating the $(\lambda^f.r)$ term), and the potential cost (the cost of applying a destruction). The inferred, upfront cost, as is seen in figure 5.25, is 1 more than the upfront cost of r itself. As the actual upfront cost, according to the operational semantics, as can be seen in figure 3.32, is 1 more than the cost of evaluating the term r. By the induction hypothesis, the upfront cost incurred by the operational semantics is bounded by the upfront inferred cost. The potential cost, which is the cost after a destruction is performed, is equal to the cost of r. Thus, by the induction hypothesis, the potential cost is bounded by the inferred cost.

□

This lemma demonstrates the correctness of the bounds inferred by the given rules of inference.

5.3 Polynomial time constraint

In this section we discuss the equivalency between $\mathcal{P}$, the class of decision problems which can be decided by a Deterministic Turing Machine in time polynomial with respect to the size of their input, and the class of functions that can be described in Pola. That Pola is a superset of $\mathcal{P}$ is of particular importance because it affects the expressiveness of the language, as described further in chapter 6.

Definition 19. *For clarity, we denote* CP *to be the set of decision problems that can be decided in Compositional Pola. Note that Compositional Pola is a language which deals with functions, not only decision problems, but to form a relationship with the class of decision problems $\mathcal{P}$, we limit* CP *to functions which yield a boolean value, thus allowing a direct comparison with decision problems.*

Proposition 3. $\mathsf{CP} = \mathcal{P}$.

Proof. We show $\mathsf{CP} \supseteq \mathcal{P}$. For any problem p in $\mathcal{P}$, there is a Deterministic Turing Machine M which computes p in time $q(x)$ for some polynomial q, relative to input size x. Without loss of generality, we assume M contains three tape symbols (zero, one, blank). From M we can construct a Compositional Pola function `transition` which has two player parameters: the current tape symbol and the current state, and returns the next state, direction to move the tape head and the new tape symbol. From this `transition` function, we can construct a Compositional Pola function `step` which has 1 opponent parameter (the beginning of the work tape) and 2 player parameters (the current state, and the tail of the work tape) and returns the new state, the new beginning of the work tape, and the new tail of the work tape. We also define a Compositional Pola function `accepting`, which has one player parameter (the current state) and returns a

value of true or false indicating whether the given state is an accepting state. We can also construct a Compositional Pola function `q` which takes an opponent parameter, a natural number, representing the length of the input on the tape, and returns a natural number which is greater than or equal to the number of steps needed to compute M to a halting state. Then, for input `input`, we can construct a **fold** construct which recurses $q(length(input))$ times, each time applying the *step* function. The `accepting` function is applied to the final state in the `fold` construct to decide if the input is accepted. This successfully simulates the Turing Machine M to a halting state, as can be seen in figure 5.28. Hence, $\mathsf{CP} \supseteq \mathcal{P}$.

We show $\mathsf{CP} \subseteq \mathcal{P}$. Consider some Compositional Function, $f \in \mathsf{CP}$, which is well-typed, and thus has an inferred time bound. From lemma 3, we can see that every time bound inferred by the rules of time bound inference given in section 5.2 correctly bound from above the number of steps needed to evaluate f via the operational semantics. Thus, the number of steps required to evaluate f is a polynomial with respect to the size of its input. We construct a Deterministic Turing Machine, M, which simulates the function via Compositional Pola's operational semantics. The size of the variable environment in Compositional Pola is bounded by the number of steps needed to complete the evaluation, according to the operational semantics. As the time required by M to simulate the operational semantics of the language is bounded by the size of the environment, and the number of steps in the operational semantics is polynomial with respect to the size of the input, M must necessarily halt in polynomial time and thus is deciding a problem in $\mathcal{P}$. Hence, $\mathsf{CP} \subseteq \mathcal{P}$.

Thus, the class of algorithms which can be expressed in Compositional Pola is equivalent to $\mathcal{P}$. □

```
1  data TapeSymbol → c
2     = Blank : () → c
3     | Marked : () → c;
4  data TapeDirection → c
5     = L : () → c
6     | R : () → c;
7  length = x |  .fold f(x) as {
8     Nil.Zero;
9     Cons(−, − | t).Succ(f(t)) }
10 in f(x);
11 append = n | l, v.fold f(n, l) as {
12    Zero.Cons(v, Nil);
13    Succ(− | n).peek l of {
14       Nil.Cons(v, Nil);
15       Cons(x, − | xs).Cons(x, f(n, xs)) } }
16 in f(n, l);
17 removeLast = n | l.fold f(n, l) as {
18    Zero.Nil;
19    Succ(− | n).peek l of {
20       Nil.Nil;
21       Cons(x, xs).peek xs of {
22          Nil.Nil;
23          Cons(−, −).Cons(x, f(n, xs)) } } }
24 in f(n, l);
25 first =  | l.peek l of {
26    Nil.Blank;
27    Cons(x, −).x };
28 removeFirst =  | l.peek l of {
29    Nil.Nil;
30    Cons(−, l′).l′ };
31 step = n | s, h, t.peek transition(s, first(t)) of {
32    (s′, v, L).(s′, append(n, h, v), removeFirst(t));
33    (s′, v, R).(s′, removeLast(n, h), Cons(v, t)) };
34 sim = input |  .fold f(n, s, h, t) as {
35    Zero.accepting(s);
36    Succ(− | n).peek step(q(length(input)), h, s, t) of {
37       (s′, h′, t′).f(n, s′, h′, t′) } }
38 in f(q(length(input)), Zero, Nil, input);
```

Figure 5.28: A simulation of a Deterministic Turing Machine in Pola. This depends on the functions `step`, `accepting` and `q` to be defined appropriately for the given Turing Machine.

Chapter 6

Expressiveness

6.1 Expressing simple functions

In this section we explore simple functions which have historically been very difficult to express in a natural way in a constrained programming language. We show that the typing constraints necessary to provide polynomial-time bounds still allow high levels of expressiveness in the language.

Cobham's initial description of a primitive recursive programming language disallowed the implementation of an efficient less-than-or-equal-to function, *leq*. Specifically, Cobham's model allowed implementation of *leq* in time complexity $\Omega(n^2)$ even though a general recursive model would yield *leq* in time complexity $\mathcal{O}(\min(m, n))$. More recent models have addressed this issue and Pola is no exception. Figure 6.1 gives a linear-time implementation of *leq* in Pola. Figure 6.8 gives a derivation of the time bounds of *leq*, specifically $3x.\mathsf{Zero} + 4x.\mathsf{Succ} + 3$. I.e., for natural numbers n, m, we can compute *leq* in time $\mathcal{O}(\min(m, n))$, which is optimal when using Peano arithmetic.

Beyond the *leq* function, we can consider the expression of another simple function as a means to study the expressive power of Compositional Pola, insertion sort. We first introduce the *insert* function, which inserts a natural number into an already sorted list of natural numbers. Previous polynomial-time systems have been incapable of expressing insertion sort in a natural way [20]. Figure 6.9 gives an example *insert* function in Pola. Note that in Pola, this function requires the use of an auxiliary "driver" variable, d. The reason we require two opponent variables, d and e, in *insert* is that we need one opponent variable to drive the recursion (d) and one separate opponent variable to drive the *leq* function (e).

In the interest of brevity, we won't consider a full derivation of the time bounds inference for the *insert* function. We see that the function consists of one **fold** about the d variable, with each recursion invoking $leq(e, x_3)$ in the worst case. We see, then, that we will infer time bounds for *insert* in the order of $\mathcal{O}(de)$, as expected.

Once *insert* has been defined, *insertionSort* is a simple function to repeatedly insert. The Pola definition of insertion sort is given in figure 6.10. The inferred upper time bound for *insertionSort* is $\mathcal{O}(x.\mathsf{Cons}^2\, x.\mathsf{Cons}.0.\mathsf{Succ})$, i.e., quadratic in the number of elements in the list and linear in the magnitude of the largest element. In a practical system which

$$
\begin{array}{c}
\mathit{leq} = \lambda^{o} x.\lambda^{p} y.(\mathbf{fold}\ (\lambda^{f} f.\lambda^{p} x.\lambda^{p} y.\mathbf{fcase}\ x \\
(\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{True}\ () \\
+\ \lambda^{\mathsf{C}}\mathsf{Succ}.\lozenge.\lambda^{op} x_4\, x_5.\triangledown.\mathbf{peek}\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{False}\ () + \lambda^{\mathsf{C}}\mathsf{Succ}.\lozenge.\lambda^{p} x_2.\triangledown.(f\ x_5\ x_2))\ y))) \\
(x)_{\mathbf{o}}\ y
\end{array}
$$

```
1 | leq = x | y.fold f(x, y) as {
2 |     Zero.True;
3 |     Succ(x_4 | x_5).peek y of {
4 |         Zero.False;
5 |         Succ(x_2).f(x_5, x_2) } }
6 | in f(x, y);
```

Figure 6.1: An implementation of the less-than-or-equal-to function, *leq*, in Compositional Pola and Pola.

implemented natural numbers as machine words, *leq* would be a constant-time operator, and we would get the expected bound of $\mathcal{O}(x.\mathsf{Cons}^2)$.

$$\dfrac{\dfrac{\dfrac{}{(f:0),(f \doteq 0) \| \cdots \mid \cdots \mathrel{\mathcal{Y}} f \doteq 0} \quad \dfrac{}{\cdots \| \cdots \mid (x_2:0),(x_5:x.\mathsf{Succ}),(y:y) \mathrel{\mathcal{Y}} x_5 \doteq 1}}{(f:0),(f \doteq 0) \| (x:x),(x_4:x.\mathsf{Succ}) \mid (x_2:0),(x_5:x.\mathsf{Succ}),(y:y) \mathrel{\mathcal{Y}} f\ x_5 \doteq 1} \quad \dfrac{}{\cdots \| \cdots \mid \begin{array}{c}(x_2:0),\\(x_5:x.\mathsf{Succ}),\\(y:y)\end{array} \mathrel{\mathcal{Y}} x_2 \doteq 1}}{(f:0),(f \doteq 0) \| (x:x),(x_4:x.\mathsf{Succ}) \mid (x_2:0),(x_5:x.\mathsf{Succ}),(y:y) \mathrel{\mathcal{Y}} f\ x_5\ x_2 \doteq 2}$$

Figure 6.2: A continuation of the derivation given below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdots \| \cdots \mid \cdots \mathrel{\mathcal{Y}} \mathsf{False} \doteq 1} \quad \dfrac{}{\cdots \| \cdots \mid \cdots \mathrel{\mathcal{Y}} () \doteq 1}}{(f:0),(f \doteq 0) \| \cdots \mid \cdots \mathrel{\mathcal{Y}} \mathsf{False}\ () \doteq 3}}{\begin{array}{c}(f:0),\\(f \doteq 0)\end{array} \| \cdots \mid \begin{array}{c}(C:0),(N:0),\\\cdots\end{array} \mathrel{\mathcal{Y}} \triangledown.\mathsf{False}\ () \doteq 3}}{\begin{array}{c}(f:0),\\(f \doteq 0)\end{array} \| \cdots \mid \begin{array}{c}(C:0),\\\cdots\end{array} \mathrel{\mathcal{Y}} \lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{False}\ () \doteq \langle \mathsf{Zero}:3 \rangle}}{(f:0),(f \doteq 0) \| \begin{array}{c}(x:x),\\(x_4:x.\mathsf{Succ})\end{array} \mid \begin{array}{c}(C:0),\\(x_5:x.\mathsf{Succ}),(y:y)\end{array} \mathrel{\mathcal{Y}} \lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{False}\ () + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^p x_2.\triangledown.(f\ x_5\ x_2) \doteq \langle \mathsf{Zero}:3 \mid \mathsf{Succ}:2 \rangle}$$

Continued in figure 6.2

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(f:0),(f \doteq 0) \| \cdots \mid \begin{array}{c}(C:0),(N:0),\\(x_2:0),(x_5:x.\mathsf{Succ}),\\(y:y)\end{array} \mathrel{\mathcal{Y}} \triangledown.(f\ x_5\ x_2) \doteq 2}}{(f:0),(f \doteq 0) \| \cdots \mid \begin{array}{c}(C:0),(N:0),\\(x_5:x.\mathsf{Succ}),\\(y:y)\end{array} \mathrel{\mathcal{Y}} \lambda^p x_2.\triangledown.(f\ x_5\ x_2) \doteq 2}}{(f:0),(f \doteq 0) \| \cdots \mid \begin{array}{c}(C:0),(N:0),\\(x_5:x.\mathsf{Succ}),\\(y:y)\end{array} \mathrel{\mathcal{Y}} \Diamond.\lambda^p x_2.\triangledown.(f\ x_5\ x_2) \doteq 2}}{\begin{array}{c}(f:0),\\(f \doteq 0)\end{array} \| \cdots \mid \begin{array}{c}(C:0),\\(x_5:x.\mathsf{Succ}),\\(y:y)\end{array} \mathrel{\mathcal{Y}} \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^p x_2.\triangledown.(f\ x_5\ x_2) \doteq \langle \mathsf{Succ}:2 \rangle}$$

Figure 6.3: A continuation of the derivation given below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(f:0),(f \doteq 0) \| (x:x),(x_4:x.\mathsf{Succ}) \mid (x_5:x.\mathsf{Succ}),(y:y) \succcurlyeq y \doteq 1}\ \text{Continued in figure 6.3}}{(f:0),(f \doteq 0) \parallel (x:x),(x_4:x.\mathsf{Succ}) \mid (x_5:x.\mathsf{Succ}),(y:y) \succcurlyeq \begin{array}{c}\mathbf{peek}\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{False}\ () \\ +\ \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{p}x_2.\triangledown.(f\ x_5\ x_2))\ y\end{array} \doteq 1 + \max\langle \mathsf{Zero}:3 \mid \mathsf{Succ}:2\rangle = 4}}{(f:0),(f \doteq 0) \| \begin{array}{c}(x:x),\\(x_4:x.\mathsf{Succ})\end{array} \mid \begin{array}{c}(C:x.\mathsf{Succ}),(N:0),\\(x_5:x.\mathsf{Succ}),(y:y)\end{array} \succcurlyeq \triangledown.\mathbf{peek}\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{False}\ () + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{p}x_2.\triangledown.(f\ x_5\ x_2))\ y \doteq 4}}{(f:0),(f \doteq 0) \| (x:x) \mid (C:x.\mathsf{Succ}),(N:0),(y:y) \succcurlyeq \lambda^{op}x_4\,x_5.\triangledown.\mathbf{peek}\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{False}\ () + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{p}x_2.\triangledown.(f\ x_5\ x_2))\ y \doteq 4}}{(f:0),(f \doteq 0) \| (x:x) \mid (C:x.\mathsf{Succ}),(N:0),(y:y) \succcurlyeq \Diamond.\lambda^{op}x_4\,x_5.\triangledown.\mathbf{peek}\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{False}\ () + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{p}x_2.\triangledown.(f\ x_5\ x_2))\ y \doteq 4}}{(f:0),(f \doteq 0) \| (x:x) \mid (C:x),(y:y) \succcurlyeq \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op}x_4\,x_5.\triangledown.\mathbf{peek}\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{False}\ () + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{p}x_2.\triangledown.(f\ x_5\ x_2))\ y \doteq \langle \mathsf{Succ}:4\rangle}$$

Figure 6.4: Continuation of the derivation given below.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(f:0),(f \doteq 0) \| (x:x) \mid (y:y) \succcurlyeq \mathsf{True} \doteq 1}\quad \dfrac{}{(f:0),(f \doteq 0) \| (x:x) \mid (y:y) \succcurlyeq () \doteq 1}}{(f:0),(f \doteq 0) \| (x:x) \mid (y:y) \succcurlyeq \mathsf{True}\ () \doteq 3}}{(f:0),(f \doteq 0) \| (x:x) \mid (C:x.\mathsf{Zero}),(N:0),(y:y) \succcurlyeq \triangledown.\mathsf{True}\ () \doteq 3}}{(f:0),(f \doteq 0) \| (x:x) \mid (C:x),(y:y) \succcurlyeq \lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{True}\ () \doteq \langle \mathsf{Zero}:3\rangle}\ \text{Continued in figure 6.4}}{(f:0),(f \doteq 0) \| (x:x) \mid (C:x),(y:y) \succcurlyeq \begin{array}{c}\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{True}\ () \\ +\ \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op}x_4\,x_5.\triangledown.\mathbf{peek} \\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{False}\ () + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{p}x_2.\triangledown.(f\ x_5\ x_2)) \\ y\end{array} \doteq \langle \mathsf{Zero}:3 \mid \mathsf{Succ}:4\rangle}$$

Figure 6.5: Continuation of the derivation given below.

$\dfrac{}{(f:0),(f \doteq 0) \| (x:x) \mid (y:y) \vdash x : x}$ $\dfrac{}{(f:0),(f \doteq 0) \| (x:x) \mid (y:y) \succcurlyeq x \doteq 1}$ Continued in figure 6.5

$$\frac{}{\begin{array}{l}(f:0),\\(f \doteq 0)\end{array} \quad \| (x:x) \mid (y:y) \succcurlyeq \quad \begin{array}{c}\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{True}\ ()\\ +\ \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_4\, x_5.\triangledown.\mathbf{peek}\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{False}\ () + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{p} x_2.\triangledown.(f\ x_5\ x_2))\ y)\end{array} \quad \doteq 3x.\mathsf{Zero} + 4x.\mathsf{Succ} + 1}$$

$$\frac{}{\begin{array}{l}(f:0),\\(f \doteq 0)\end{array} \quad \| (x:x) \mid (y:y) \succcurlyeq \quad \begin{array}{c}\lambda^{p} y.\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{True}\ ()\\ +\ \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_4\, x_5.\triangledown.\mathbf{peek}\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{False}\ () + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{p} x_2.\triangledown.(f\ x_5\ x_2))\ y)\end{array} \quad \doteq 3x.\mathsf{Zero} + 4x.\mathsf{Succ} + 1}$$

$$\frac{}{(f:0),(f \doteq 0) \| (x:x) \mid (y:y) \succcurlyeq \quad \begin{array}{c}\lambda^{p} x.\lambda^{p} y.\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{True}\ ()\\ +\ \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_4\, x_5.\triangledown.\mathbf{peek}\\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{False}\ () + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{p} x_2.\triangledown.(f\ x_5\ x_2))\ y)\end{array} \quad \doteq x \to 3x.\mathsf{Zero} + 4x.\mathsf{Succ} + 1}$$

$$\frac{}{\| (x:x) \mid (y:y) \succcurlyeq \quad \begin{array}{c}\lambda^{f} f.\lambda^{p} x.\lambda^{p} y.\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{True}\ ()\\ +\ \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_4\, x_5.\triangledown.\mathbf{peek}\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{False}\ () + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{p} x_2.\triangledown.(f\ x_5\ x_2))\ y)\end{array} \quad \doteq x \to 3x.\mathsf{Zero} + 4x.\mathsf{Succ} + 1}$$

Figure 6.6: A continuation of the derivation given below.

Continued in figure 6.6 $\dfrac{}{\| (x:x) \mid (y:y) \succcurlyeq x \doteq 1}$ $\dfrac{}{\| (x:x) \mid (y:y) \vdash x : x}$

$$\frac{}{\| (x:x) \mid (y:y) \succcurlyeq \quad \begin{array}{c}(\mathbf{fold}\ (\lambda^{f} f.\lambda^{p} x.\lambda^{p} y.\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{True}\ ()\\ +\ \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_4\, x_5.\triangledown.\mathbf{peek}\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{False}\ () + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{p} x_2.\triangledown.(f\ x_5\ x_2))\ y)))\\ (x)_{\mathbf{o}}\end{array} \quad \doteq 3x.\mathsf{Zero} + 4x.\mathsf{Succ} + 2}$$

Figure 6.7: A continuation of the derivation given below.

Continued in figure 6.7 $\quad \overline{\|(x : x) \mid (y : y) \succcurlyeq y \doteq 1}$

$$\|(x : x) \mid (y : y) \succcurlyeq \begin{array}{c}(\mathbf{fold}\ (\lambda^f f.\lambda^p x.\lambda^p y.\mathbf{fcase}\ x \\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{True}\ () \\ +\ \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^{op} x_4\, x_5.\triangledown.\mathbf{peek}\ (\lambda^{\mathsf{C}}\mathsf{Zero}.\triangledown.\mathsf{False}\ () + \lambda^{\mathsf{C}}\mathsf{Succ}.\Diamond.\lambda^p x_2.\triangledown.(f\ x_5\ x_2))\ y))) \\ (x)_{\mathbf{o}}\ y\end{array} \doteq 3x.\mathsf{Zero} + 4x.\mathsf{Succ} + 3$$

Figure 6.8: Time bound inference for the *leq* function given in figure 6.1.

$$
\begin{array}{c}
\mathit{insert} = \lambda^o d.\lambda^o e.\lambda^p l.(\lambda^f f.(f\ d\ l))\ (\mathbf{fold} \\
(\lambda^f f.\lambda^p d.\lambda^p l.\mathbf{fcase}\ d \\
(\lambda^{\mathsf{C}}\mathsf{Nil}.\triangledown.\mathsf{Cons}\ (e \times (\mathsf{Nil}\ ()) \times ()) + \\
\lambda^{\mathsf{C}}\mathsf{Cons}.\Diamond.\lambda^{op} x_7\, x_9.\Diamond.\lambda^{op} x_8\, x_1.\triangledown.\mathbf{peek} \\
(\lambda^{\mathsf{C}}\mathsf{Nil}.\triangledown.\mathsf{Cons}\ (e \times (\mathsf{Nil}\ ()) \times ()) + \\
\lambda^{\mathsf{C}}\mathsf{Cons}.\Diamond.\lambda^{p} x_5.\Diamond.\lambda^{p} x_6.\triangledown.(\lambda^{p} \mathit{subject}_2.\mathbf{peek} \\
(\lambda^{\mathsf{C}}\mathsf{False}.\triangledown.\mathsf{Cons}\ (x_3 \times ((f\ x_1\ x_4)) \times ()) + \\
\lambda^{\mathsf{C}}\mathsf{True}.\triangledown.\mathsf{Cons}\ (e \times (\mathsf{Cons}\ (x_3 \times x_4 \times ())) \times ()))\ \mathit{subject}_2)\ ((\mathit{leq}\ e\ x_3)))\ l)))
\end{array}
$$

```
1 | insert = d, e | l.fold f(d, l) as {
2 |     Nil.Cons(e, Nil);
3 |     Cons(x7 | x9, x8 | x1).peek l of {
4 |         Nil.Cons(e, Nil);
5 |         Cons(x3 | x5, x4 | x6).peek subject2 of {
6 |                 False.Cons(x3, f(x1, x4));
7 |                 True.Cons(e, Cons(x3, x4)) }
8 |             where subject2 = leq(e, x3) } }
9 | in f(d, l);
```

Figure 6.9: An implementation of the insert function, *insert*, in Compositional Pola and Pola.

6.2 Limits

We have shown two historically important functions, *leq* and *insertionSort*, can be written in a natural style in Pola and have tight bounds inferred. Where Pola fails in its expressiveness is in the "divide and conquer" algorithms. The typing system of Pola is not amenable to breaking up data structures, aside from how they're explicitly structured. For example, one cannot split a list in two pieces in Pola and then recurse separately on each half of the list, due to restrictions on the number of recursive calls. This implies that one cannot write *mergeSort* in a natural style.

6.3 Variations of Pola

So far, we have discussed Compositional Pola and Pola, two related languages which allow functions equal to the polynomial-time functions. In this section, we have two interests: to consider variations that allow different complexity constraints; and to consider variations of Pola that allow more programming expressiveness without compromising the restriction to polynomial time.

6.3.1 An elementary-recursive language

Consider a variant of Compositional Pola that allows full duplication in the player world, which we will call Affine-Free Compositional Pola. Figure 6.11 shows the modification to the type inference rule for the **peek** construct (originally in figure 3.34 for Affine-Free

$$insertionSort = \lambda^o x.(\lambda^f f.(f\ x\ (\mathsf{Nil}\ ()))))\ (\mathbf{fold}\ (\lambda^f f.\lambda^p x.\lambda^p acc.\mathbf{fcase}\ x\ (\lambda^{\mathsf{C}}\mathsf{Nil}.\triangledown.acc + \lambda^{\mathsf{C}}\mathsf{Cons}.\diamondsuit.\lambda^o x_1.\diamondsuit.\lambda^{op} x_2\, x_4.\triangledown.(f\ x_4\ ((insert\ x\ x_1\ acc))))))$$

```
1 | insertionSort = x |  .fold f(−, acc) as {
2 |     Nil.acc;
3 |     Cons(x1, x2 | x4).f(x4, insert(x, x1, acc)) }
4 | in f(x, Nil);
```

Figure 6.10: An implementation of the insertion sort function, *insertionSort*, in Compositional Pola and Pola.

$$\text{Peek in Compositional Pola} \qquad \frac{\Xi \| \Gamma \mid \Delta_1 \vdash t : \beta \quad \Xi \| \Gamma \mid \Delta_2 \vdash b : \gamma}{\Xi \| \Gamma \mid \Delta_1, \Delta_2 \vdash \mathbf{peek}\ b\ t : \alpha}$$

$$\text{Peek in Affine-Free Compositional Pola} \qquad \frac{\Xi \| \Gamma \mid \Delta \vdash t : \beta \quad \Xi \| \Gamma \mid \Delta \vdash b : \gamma}{\Xi \| \Gamma \mid \Delta \vdash \mathbf{peek}\ b\ t : \alpha}$$

Figure 6.11: A comparison of type inference rules for the **peek** construct in Compositional Pola (top) and Affine-Free Compositional Pola (bottom).

Compositional Pola. Although figure 3.34 shows only one such rule, *all* type inference constructs that rely on the split between player context Δ_1, Δ_2 in section 3.6 would be modified to exclude this split. In effect, this means that there is no restriction on the number of occurrences of any player variable.

We can quickly see that this modification allows computations beyond polynomial time. Figure 6.12 gives a Affine-Free Compositional Pola program which defines the *exp2* function, a function which will compute 2^x for any natural number x. Because each natural number, x, in Affine-Free Compositional Pola is represented via x applications of the Succ constructor, necessarily computational time is at least 2^x according to the operational semantics, and thus Affine-Free Compositional Pola allows super-polynomial computation.

The complexity class of elementary-recursive, written $\mathcal{EL}$, is the set of decision problems which can be decided in time bounded by the exponential hierarchy, i.e., $2^n \cup 2^{2^n} \cup 2^{2^{2^n}} \cup \cdots$. It is a strict subset of the Primitive Recursive class studied in section 2.2.1. The set of functions computable by an Affine-Free Compositional Pola program will hereafter be denoted AFCP.

Proposition 4. $\mathsf{AFCP} = \mathcal{EL}$.

Proof. This proof requires two parts, to show equivalence.

1. $\mathsf{AFCP} \supseteq \mathcal{EL}$.

   We show this by demonstrating that any decision problem in $\mathcal{EL}$ can be decided in time $f(x)$, for some function $f(x)$, within the exponential hierarchy. Further, the function $f(x)$ can be computed in AFCP.

```
1  | data BinaryTree → c
2  |     = Leaf :→ c
3  |     | Node : c, c → c;
4  | genTree = x |  .fold f(x) as {
5  |     Zero.Leaf;
6  |     Succ(− | n).Node(f(n), f(n)) }
7  | in f(x);
8  | countTree = t |  .fold f(t, s) as {
9  |     Leaf.Succ(s);
10 |     Node(− | t, − | u).f(t, f(u, Succ(s))) }
11 | in f(t);
12 | exp2 = x |  .countTree(genTree(x));
```

Figure 6.12: An example of exponential-time behaviour in Affine-Free Compositional Pola.

If $f(x) \leq c^x$—i.e., $f(x)$ is a simple exponential, we can use the AFCP code in figure 6.12 to compute $f(x)$. If $c = 2$, the function can be used as-is. In the case that $c > 2$, we can modify the definition of the Node constructor to have c subterms instead of 2 subterms.

If $f(x) \leq c^{g(x)}$ where $g(x)$ is a function in the exponential hierarchy, we can assume there is a AFCP function g which computes $g(x)$. As in figure 6.12, we can then construct a function *exp2* which computes 2^x and evaluate the AFCP term $exp2(g(n))$ to compute $f(x)$. In the case that $c > 2$, we can again modify the Node constructor to set the number of subterms accordingly.

As Affine-Free Compositional Pola can compute any function in the exponential hierarchy, it can simulate any Turing machine which halts in a time bounded by a function in the exponentional hierarchy, and thus AFCP $\supseteq \mathcal{EL}$.

2. AFCP $\subseteq \mathcal{EL}$.

   Similarly to lemma 2, we can see the number of invocations of a recursive function f is constant in Affine-Free Compositional Pola per constructor of the term being folded over. Further, because each recursive call necessarily requires a player variable, namely the first argument to the recursive function f, of universal type, it is not possible to use the result of a recursive call to drive another recursion.

   Because the number of invocations to a recursive function is a constant, $c \in \mathbb{N}$, the total computational cost of evaluation a fold in Affine-Free Compositional Pola must be within $\mathcal{O}(c^{g(n)})$ where n is the number of constructors within the term being folded over and $g(n)$ is the computational cost of the body of the **fold** construct, ignoring invocations to the recursive function.

   Because nested **fold** constructs can be used, $g(n)$ is itself in the exponential hierarchy, and consequently the cost of evaluating a **fold** must be in the exponential hierarchy and consequently halt within elementary time.

Since $\mathsf{AFCP} \subseteq \mathcal{EL}$ and $\mathsf{AFCP} \supseteq \mathcal{EL}$, it follows that $\mathsf{AFCP} = \mathcal{EL}$. □

One interesting note of the result of proposition 4 is that it is distinguished from other "categorical" programming languages, such as Charity described in section 2.2.4. Charity is also strictly recursive—all well-typed programs must necessarily halt—and also uses folds on inductive types and unfolds on coinductive types. However, Ackermann's function can be written in Charity, making it strictly more powerful than primitive recursive while still being recursive, whereas Ackermann's function is impossible to write in Affine-Free Compositional Pola.

The distinction between the two in terms of computational power can be seen in the division between opponent variables and player variables, a division which does not exist in Charity. In Charity one can freely use the result of a fold as an argument to an unfold, or use the result of a recursive function call to drive a fold, two things which are forbidden in Affine-Free Compositional Pola, which gives Charity more computational power.

6.3.2 A PSPACE language

We will now consider a restrained version of Affine-Free Compositional Pola, called Exploratory Compositional Pola, hereafter denoted ECP. Rather than allow unrestricted duplication of player variables, we will allow duplication of player variables only in one context: the **peek** construct. The typing restriction on the new **peek** construct is the same as is shown in figure 3.34. Of note, compared to Pola, the player context (Δ) in ECP may be duplicated between the subject of the **peek** and the body of the **peek**.

We add one further typing restriction, which is that every inductive type must have at least one constructor which is not recursive. In practice, this is not a serious restriction, as an object of an inductive type without a non-recursive constructor could never be constructed.

Since the **peek** is the only context in which duplication is permitted, the style of exponential functions given in figure 6.12 is not permitted. However, computational power beyond that of polynomial-time functions is permitted.

A solution to the Quantified Boolean Formula Problem is given in figure 6.13. The Quantified Boolean Formula problem determines a solution to a Boolean formula, Φ, where Φ, has one of the following forms:

- $\Phi \equiv x$ (a variable);
- $\Phi \equiv \Phi \wedge \Phi$ (conjunction);
- $\Phi \equiv \Phi \vee \Phi$ (disjunction);
- $\Phi \equiv \neg\Phi$ (negation);
- $\Phi \equiv \exists x.\Phi$ (existentially quantified);
- $\Phi \equiv \forall x.\Phi$ (universally quantified).

```
1  | data QBF → c
2  |    = Variable : Nat → c
3  |    | And : c, c → c
4  |    | Or : c, c → c
5  |    | Not : c → c
6  |    | Exists : c → c
7  |    | Forall : c → c;
8  | head =  | list.peek list of { Cons(x, −).x };
9  | tail =  | list.peek list of { Cons(−, xs).xs };
10 | lookup = var | env.fold f(x, env) as {
11 |    Zero.head(env);
12 |    Succ(− | x).f(x, tail(env)) }
13 | in f(var, env);
14 | eval = eq | env.fold f(x, env) as {
15 |    Variable(x).lookup(x, env);
16 |    And(− | a, − | b).peek f(a, env) of {
17 |          False.False;
18 |          True.f(b, env) }
19 |    Or(− | a, − | b).peek f(a, env) of {
20 |          False.f(b, env);
21 |          True.True }
22 |    Not(− | a).peek f(a, env) of {
23 |          False.True;
24 |          True.False }
25 |    Exists(− | a).peek f(a, Cons(False, env)) of {
26 |          False.f(a, Cons(True, env));
27 |          True.True }
28 |    Forall(− | a).peek f(a, Cons(False, env)) of {
29 |          False.False;
30 |          True.f(a, Cons(True, env)) }
31 | in f(eq, env);
```

Figure 6.13: A solution to the Quantified Boolean Formula problem, given in Exploratory Compositional Pola.

In the solution given in figure 6.13, we take a slightly modified version of quantified formulas, which is to represent them in point-free notation. Rather than represent variables symbolically, we represent them as a natural number indicating the distance to the quantifier in which they were introduced. The *distance* indicates the number of other quantifiers that have been introduced after the intended quantifier in the same scope. For example, the formula $\forall x.\exists y.y \wedge (\forall z.(\exists w.w \wedge z) \vee x)$ would be given as $\forall.\exists.0 \wedge (\forall.(\exists.0 \wedge 1) \vee 2)$ in point-free notation.

Proposition 5. *Exploratory Compositional Pola is complete in polynomial space. I.e.,* $\mathsf{ECP} \supseteq$ *PSPACE.*

Proof. Figure 6.13 gives a solution to the Quantified Boolean Formula problem. The Quantified Boolean Formula problem is complete in PSPACE. □

To complete the proof that Exploratory Compositional Pola is equal to PSPACE, we now prove soundness. To do this, we will need to make a transformation on ECP programs.

Lemma 4. *For any Exploratory Compositional Pola term* **peek** *t b, there exists a Compositional Pola term* t' *that is evaluated in constant time such that* **peek** t' *b is well-typed.*

Proof. Because of the typing restrictions on the **peek** construct, t is necessarily an inductive type. Because every inductive type in an ECP program necessarily has at least one non-recursive constructor, C_t, replacing t with C_t will not change the type of the **peek** body b. Further, since the constructor is not recursive, there necessarily exists some term x which evaluates in constant time such that $\mathsf{C}_t\ x$ is well-typed. $t' = \mathsf{C}_t\ x$. □

Following from lemma 4, we define a transform $p \rightarrow_\theta p'$, where p is an Exploratory Compositional Pola program and p' is a Compositional Pola program with exactly the same structure of p' except that the subjects of all peeks are stubbed out, according to lemma 4. p' is necessarily a well-typed Compositional Pola program.

Proposition 6. *Every function in Exploratory Compositional Pola consumes at most polynomial space. I.e.,* $\mathsf{ECP} \subseteq$ *PSPACE.*

Proof. We rely on the fact that every Alternating Turing Machine which halts in polynomial time consumes at most polynomial space [3], and thus it suffices to show that every function in Exploratory Compositional Pola can be simulated via an Alternating Turing Machine in polynomial time.

An Alternating Turing Machine is a non-deterministic Turing Machine in which every state is labelled as a $\wedge$ states or a $\vee$ state. States labelled as $\vee$ states must have at least one non-deterministic path reach an accept state; states labelled as a $\wedge$ state must have every non-deterministic path reach an accept state. Deterministic states can be labelled as either without any change in semantics.

Owing to proposition 3, we know that, for any Compositional Pola program, there is a Turing Machine M which simulates the program in polynomial time. Thus, for any Exploratory Compositional Pola program, p, there is a Turing Machine $M_{p'}$ which simulates the program p' in polynomial time, where $p \rightarrow_\theta p'$. As every deterministic

$$\frac{\Xi \| \Gamma \mid \Delta_1 \vdash t : \alpha \quad \Xi \| \Gamma \mid \Delta_2 \vdash b : \beta}{\Xi \| \Gamma \mid \Delta_1, \Delta_2 \vdash \mathbf{peek}\ t\ b : \beta}$$ Peek (Compositional Pola)

$$\frac{\Xi \| \Gamma \mid \Delta_1, \Delta_2 \vdash t : \alpha \quad \Xi \| \Gamma \mid \Delta_1, \Delta_3 \vdash b : \beta}{\Xi \| \Gamma \mid \Delta_1, \Delta_2, \Delta_3 \vdash \mathbf{peek}\ t\ b : \beta}\ \Delta_1 \text{ is safe}$$ Peek (Dangerous Compositional Pola)

Figure 6.14: The simplified typing rules of the **peek** construct in Compositional Pola (top) and Dangerous Compositional Pola (bottom). The introduction of variables of type α is not given in these rules, for brevity.

Turing Machine can be easily transformed to an Alternating Turing Machine, by labelling each state as a $\wedge$ state, we can say that $M_{p'}$ is an Alternating Turing Machine.

We can then transform $M_{p'}$ by replacing each **peek** construct. Each **peek** construct in $M_{p'}$ is has a representation for **peek** t' $(\mathsf{C}_1.e_1 + \cdots + \mathsf{C}_n.e_n)$ in the Alternating Turing Machine $M_{p'}$. We replace this with $(t'$ **is** $\mathsf{C}_1 \wedge \mathsf{C}_1.e_1) \vee \cdots \vee (t'$ **is** $\mathsf{C}_n \wedge \mathsf{C}_n.e_n)$ where the notation $(t'$ **is** $\mathsf{C}_i)$ indicates an evaluation of t' and an assertion that it matches constructor C_i, leading to a reject state if the assertion fails.

This constructed Turing Machine is an Alternating Turing Machine which halts in polynomial time. Thus, $\mathsf{ECP} \subseteq \mathit{PSPACE}$. □

Consequently, we can see that $\mathsf{ECP} = \mathit{PSPACE}$, and thus allowing unrestricted duplication of player variables allows computational complexity equal to allowing polynomial space.

6.3.3 A more expressive P-complete language

Finally we consider a variant of Compositional Pola, denoted Dangerous Compositional Pola (DCP), which remains constrained to polynomial-time functions, but allows more expressive power to the programmer, by selectively relaxing the restrictions on duplication in the player world.

We introduce two new types in DCP: safe and dangerous. Both are parametric types and will be denoted $\langle\alpha\rangle$ (safe type α) and $\rangle\alpha\langle$ (dangerous type α). Types of variables in the opponent context will not be designated as safe or dangerous; conversely, every variable in the player context will be designated as either safe or dangerous. Safe variables are those that may be duplicated and dangerous variables are those which could be used to cause super–polynomial-time behaviour if duplicated.

The only term which duplicates safe variables is the **peek** construct, as described in figure 6.14. The player context is divided into two sub-contexts, Δ_1, which contains only safe variables, and (Δ_2, Δ_3), which contains only dangerous variables. Δ_1 can be duplicated between the subject and the body of the **peek** construct, whereas (Δ_2, Δ_3) may not be duplicated and must be split between the subject and the body.

Almost all terms from Compositional Pola generate safe types. There are two terms which can generate dangerous types. Recursive variables—i.e., those of universal type—

```
1  | data c → Fn(a, b)
2  |    = Eval : c, a → b;
3  | both =   | rec.peek Eval[rec](False) of {
4  |          False.False;
5  |          True.Eval[rec](True) };
6  | either =   | rec.peek Eval[rec](False) of {
7  |          False.Eval[rec](True);
8  |          True.True };
9  | eval = eq | env.fold f(x, env) as {
10 |          ⋮
11 |    Exists(− | a).either(( Eval : z.f(a, Cons(z, env)) ));
12 |    Forall(− | a).both(( Eval : z.f(a, Cons(z, env)) )) }
13 | in f(eq, env);
```

Figure 6.15: An example of how to generate an exponential-time function if safe and dangerous type restrictions are not put on records.

$$\frac{}{\Xi \| \Gamma \mid \Delta \vdash \mathbf{record} : \bar{\mathbf{br}}(\alpha) \to \langle \alpha \rangle}\ \Delta \text{ is safe} \qquad \text{Record}$$

$$\frac{}{\Xi \| \Gamma \mid \Delta \vdash \mathbf{record} : \bar{\mathbf{br}}(\alpha) \to \rangle \alpha \langle}\ \Delta \text{ is dangerous} \qquad \text{Record}$$

Figure 6.16: Modified typing rules for records in Dangerous Compositional Pola.

are dangerous. Records can also be have dangerous type because they can be used to implicitly duplicate a dangerous variable. Figure 6.15 gives an example of exponential-time behaviour that could occur if dangerous records were permitted to be duplicated. It is a modification of the QSAT problem given in figure 6.13, except with slight modifications to the evaluations of determining the $\exists.\Phi$ and $\forall.\Phi$ cases, shown on lines 11 and 12. Rather than duplicate the recursive variable a explicitly, we envelop it in a record and then pass that record to the *both* or *either* function, which duplicates the record and evaluates it both with False and True. As QSAT is PSPACE-complete and PSPACE $\supset \mathcal{P}$, the function is super-polynomial.

A more subtle example of exponential-time behaviour resulting from dangerous records is given in figure 6.17. Note that, in this case, the recursive variable n is never included

```
1 | g = n |  .fold f(x, y) as {
2 |    Zero.both(y);
3 |    Succ(− | n).f(n, ( Eval : z.both(y) )) }
4 | in f(n, ( Eval : z.z ));
```

Figure 6.17: A second example of an exponential-time function if safe and dangerous type restrictions are not properly put on records.

in the constructed record and never duplicated, but running time is still exponential with respect to n. Duplication of any record constructed in the context of a dangerous variable consequently needs to be considered dangerous.

Thus it is clear that records can be considered dangerous: in DCP, we would require *both* and *either* to have type signatures of $| \langle \mathsf{Fn}(\mathsf{Bool}, \mathsf{Bool}) \rangle \to \mathsf{Bool}$ and for the records generated on lines 11 and 12 to have type signatures of $\rangle \mathsf{Fn}(\mathsf{Bool}, \mathsf{Bool}) \langle$, thus disallowing this function from type-checking correctly.

Figure 6.16 gives the modified rules of typing to account for safe records and dangerous records. A record is dangerous if and only if it has any dangerous variables in its player context at the time of its construction.

A careful reading of the typing rules in figure 3.25 for Compositional Pola shows that they disallow the use of player variables within the coda of an **unfold** construct. Figure 3.30 shows a function to demonstrate the reasoning behind this, specifically that the use of a recursive function call within the code of an **unfold** can lead to exponential-time behaviour. The function *exp* given in figure 3.30 generates the infinite list $[2^n, 2^{n+1}, 2^{n+2}\, 2^{n+3}, \ldots]$. Consequently evaluating the term $\mathsf{Head}[exp(n)]$ would compute 2^n for any natural number n.

The restriction on disallowing player variables within the coda of an **unfold** can be relaxed in Dangerous Compositional Pola to make the language less restrictive, but figure 3.30 shows why restrictions are still required. Specifically, the restriction must be made that only *safe* player variables be permitted within the coda.

With these restrictions in place, Dangerous Compositional Pola is still constrained to polynomial-time functions, with slightly more freedom and expressiveness offered to the programmer, as compared to Compositional Pola.

Chapter 7

Conclusion

We have defined a novel programming language, Pola, in which every well-typed program halts in time polynomial with respect to its input. It is a functional programming language allowing algebraic inductive or coinductive datatypes to be used during computation. A practical bounds-inference algorithm is provided. Variations on the language are considered to allow greater expressiveness or computational power.

Bibliography

[1] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.

[2] Walter S. Brainerd and Lawrence H. Landweber. *Theory of computation*. Wiley, New York, 1974.

[3] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[4] Alan Cobham. The intrinsic computational difficulty of functions. In Yehoshua Bar-Hillel, editor, *Logic, Methodology, and Philosophy of Science*, pages 24–30. Elsevier/North-Holland, 1965.

[5] Robin Cockett. Charitable thoughts. Lecture notes, 1996.

[6] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report 92/480/18, Department of Computer Science, University of Calgary, 1992.

[7] L. Colson. About primitive recursive algorithms. *Theoretical Computer Science*, 83(1):57–69, 1991.

[8] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, 1977.

[9] Luis Damas and Robin Milner. Principle type-schemes for functional languages. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on the Principles of Programming Languages*, pages 207–212, 1982.

[10] Edsger Dijkstra. Recursive programming. *Numerische Mathematik*, 2(1):312–318, 1960.

[11] Edsger Dijkstra. Letters to the editor: go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[12] Maarten M. Fokkinga, Johan Jeuring, Lambert Meertens, and Erik Meijer. A translation from attribute grammars to catamorphisms. *The Squiggolist*, 2(1):20–26, 1994.

[13] Tom Fukushima and Charles Tuckey. *Charity User Manual*. University of Calgary, January 1996.

[14] Yuri Gurevich. Algebras of feasible functions. In *Proceedings of the Symposium on Foundations of Computer Science*, pages 210–214, Tucson, Arizona, 1983.

[15] Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. Towards a flow analysis for embedded system C programs. In *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 287–300, 2005.

[16] Kevin Hammond, Gudmond Grov, Greg Michaelson, and Andrew Ireland. Low-level programming in Hume: an exploration of the HW-Hume level. In *Proc. Implementation of Functional Languages*, volume 4449 of *Lecture Notes in Computer Science*, pages 91–107, Budapest, 2007.

[17] Kevin Hammond, Greg Michaelson, and Robert Pointon. Hume report, version 1.1. Webpage at http://www-fp.cs.st-andrews.ac.uk/hume/report/, 2007.

[18] Kevin Hammond, Greg Michaelson, and Pedro Vasconcelos. Bounded space programming using finite state machines and recursive functions: the Hume approach. *Transactions on Software Engineering and Methodology*, 2006.

[19] Wiebke Herding. Specification of datatypes in memory in CASL. Master's thesis, University of Edinburgh, 2004.

[20] M. Hofmann. Type systems for polynomial-time computation. Habilitation thesis. University of Darmstdat, 1999.

[21] Martin Hofmann. Programming languages capturing complexity classes. *ACM SIGACT News*, 31(1):31–42, 2000.

[22] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Annual Symposium on the Principles of Programming Languages*, pages 185–197, New Orleans, 2003.

[23] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis. In *European Symposium on Programming*, volume 3924 of *Lecture Notes in Computer Science*, pages 22–37, Vienna, 2006.

[24] Donald Knuth. *Literate Programming*. Center for the Study of Language and Information, Stanford, California, 1992.

[25] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1991.

[26] Albert M. Meyer and Dennis M. Ritchie. The complexity of loop programs. In *ACM Annual Conference*, pages 465–469, Washington, D.C., 1967.

[27] Greg Michaelson. Constraints on recursion in the Hume expression language. In *International Workshop on Implementation of Functional Programming*, Aachen, Germany, 2000.

[28] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, 17:348–375, 1978.

[29] Benjamin C. Pierce. *Types and programming languages*, chapter 22. MIT Press, 2002.

[30] Robert W. Ritchie. Classes of predictably computable functions. *Transactions of the American Mathematical Society*, 106(1):139–173, 1963.

[31] H.E. Rose. *Subrecursion: Functions and hierarchies*. Number 9 in Oxford Logic Guides. Oxford University Press, 1984.

[32] Ando Saabas and Tarmo Uustalu. A compositional natural semantics and hoare logic for low-level languages. In *In: Proceedings of the Second Workshop on Structured Operational Semantics*, pages 151–168. Elsevier, 2005.

[33] Ando Saabas and Tarmo Uustalu. Compositional type systems for stack-based low-level languages. In *Computing: The Australasian Theory Symposium*, volume 51, pages 27–39, Hobart, Australia, 2006.

[34] Marc A. Schroeder. Higher-order Charity. Master's thesis, University of Calgary, 1997.

[35] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):724–767, 1998.

[36] Mads Tofte and Jean-Pierre Talpin. Implementing the call-by-value lambda-calculus using a stack of regions. In *Proceedings of the 21st SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, Portland, Oregon, 1994. ACM Press.

Curriculum Vitae

Name:	Michael Burrell
Post-Secondary Education and Degrees:	University of Calgary Calgary, AB 2000–2004 B.Sc.H.
	University of Western Ontario London, ON 2004–2006 M.Sc.
	University of Western Ontario London, ON 2006–2017 Ph.D.
Honours and Awards:	USC Teaching Honour Roll 2009, 2011
	UWO Faculty of Science Teaching Award 2009
	SOGS Graduate Student Teaching Award 2007–2010
Related Work Experience:	Teaching Assistant The University of Western Ontario 2004–2011
	Lecturer The University of Western Ontario 2008, 2011
	Professor Sheridan College 2013–2017

Publications:

1. M.J. Burrell, J.H. Andrews, M. Daley. A useful resource bounded functional language, in: V. Geffert, J. Karhumäki, A. Bertoni, B. Preneel, P. Návrat, M. Bieliková (Eds.), 34th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM), Nový Smokovec, Slovakia, 2008, pp.198–210.

2. F. Biegler, M.J. Burrell, M. Daley. Guided insertion from RNA editing in kinetoplastids. Theoretical Computer Science, volume 387(2), 2007, pp.103–112.

3. F. Biegler, M.J. Burrell, M. Daley. Regulated RNA rewriting: modelling RNA editing with guided insertion, in: H. Leung, G. Pighizzini (Eds.), 8th International Workshop on Descriptional Complexity of Formal System, Las Cruces, NM, USA, 2006, pp.70–81.